Problem_1

May 1, 2020

0.1 Hyperplane Estimation

[1]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%config InlineBackend.figure_format = 'retina'
```

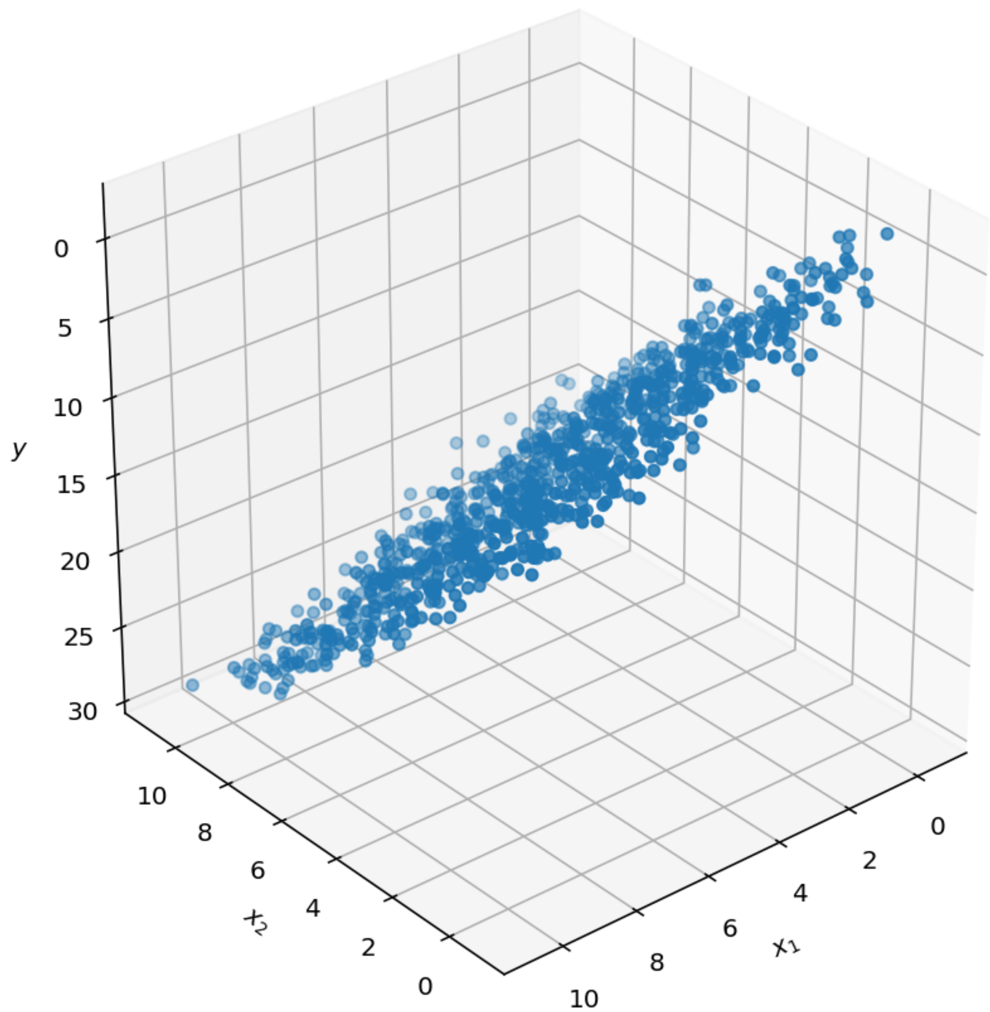
[2]:
```
X_and_Y = torch.load('hyperplane-estimation.pt')
X1 = X_and_Y[:, 0].reshape(-1, 1)     # Shape: [900, 1]
X2 = X_and_Y[:, 1].reshape(-1, 1)     # Shape: [900, 1]
Y  = X_and_Y[:, 2].reshape(-1, 1)     # Shape: [900, 1]
print(X1.shape, X2.shape, Y.shape)
```

```
torch.Size([900, 1]) torch.Size([900, 1]) torch.Size([900, 1])
```

0.1.1 Original Data

[3]:
```
# Visualization.
def vis(w0, w1, w2):
    draw_plane = (w0 is not None) and (w1 is not None) and (w2 is not None)

    if draw_plane:
        num = 30
        X_plane_range = np.linspace(0,10,num)
        X_plane_range = np.linspace(0,10,num)
        X1_plane, X2_plane = np.meshgrid(X_plane_range, X_plane_range)
        Y_plane = w0 + w1 * X1_plane + w2 * X2_plane

    fig = plt.figure(figsize = (6, 6))
    ax = Axes3D(fig, elev = -150, azim = 130)
    ax.scatter(X1.numpy(), X2.numpy(), Y.numpy())
    if draw_plane:
        ax.scatter(X1_plane, X2_plane, Y_plane)
    ax.set_xlabel('$x_1$')
    ax.set_ylabel('$x_2$')
    ax.set_zlabel('$y$')
```

```
    plt.show()

vis(None, None, None)
```



0.2 1.1

G(W) = (XW-Y)' * (XW-Y)

G(W) = W'X'XW - W'X'Y- Y'XW + Y'Y

dG(W)/dw = 2X'XW - 2 X'Y

0.3 1.2

W* = argmin(G(W))

dG(W)/dw = 2X'XW - 2 X'Y = 0

W = (X'X)^(-1) * X'Y

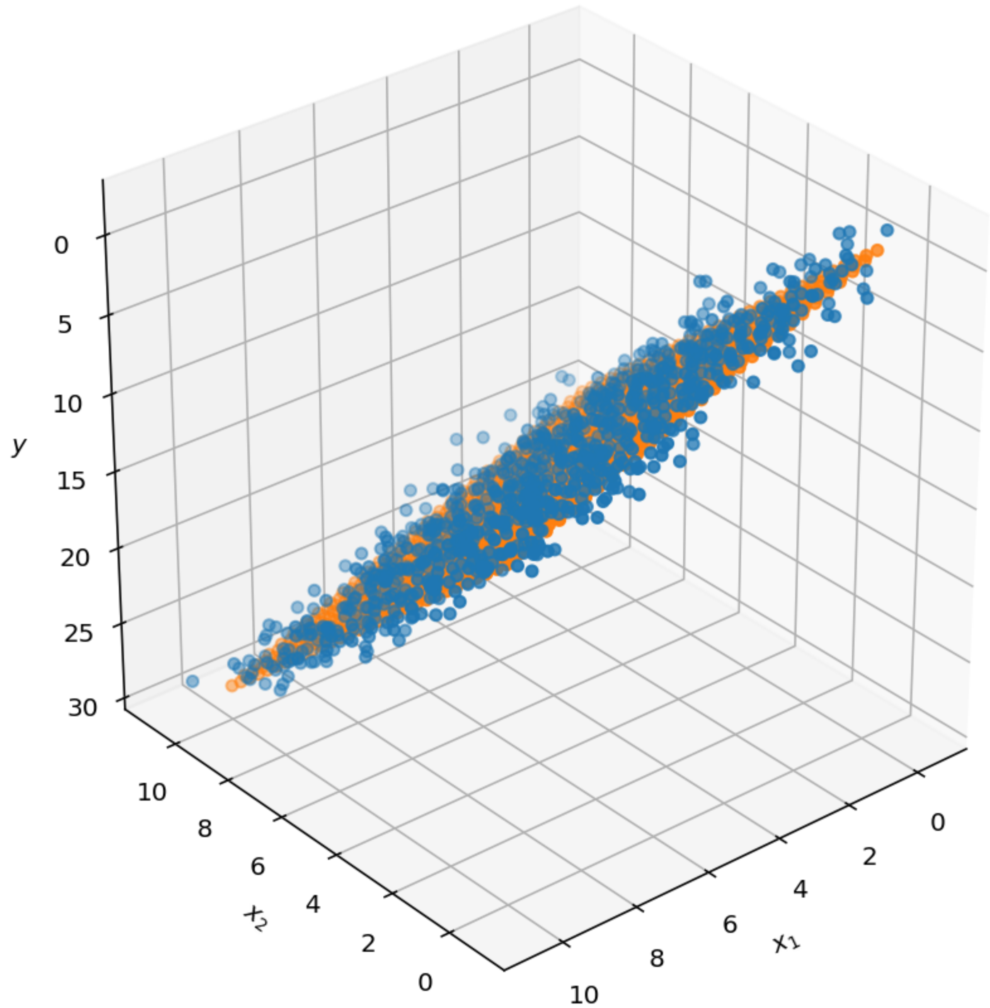
0.3.1 Hyperplane Estimation Using the Closed Form Solution

Assume data points are represented as matrices X and Y, please use the closed form solution to calculate the parameters W.

[4]:
```
X = torch.cat([torch.ones((len(X1),1)), X1, X2], dim=1)    # X contains 1, X1␣
↪and X2.
#print (X.shape)    # torch.Size([900, 3])
#print (Y.shape)    # torch.Size([900, 1])

# Compute W using the closed form solution.
A = torch.inverse(torch.mm(X.T, X))
W = torch.mm(A, torch.mm(X.T, Y)) # Hint: In the form of X and Y.
#print (W.shape)    # torch.Size([3, 1])
w0, w1, w2 = W[0,0].numpy(), W[1,0].numpy(), W[2,0].numpy()
print('y = {:.2f} + {:.2f}*x1 + {:.2f}*x2'.format(w0, w1, w2))
```

```
y = -0.70 + 0.98*x1 + 1.94*x2
```

[5]:
```
# Visualization.
vis(w0, w1, w2)
```



0.3.2 Hyperplane Estimation Using Gradient Descent

In this problem, we would like to use the gradient descent to calculate the parameters W for the hyperplane. If we have an error function (a.k.a objective function or loss function), then a typical gradient descent algorithm contains the following steps:

Step 1. Initialize the parameters W.

for i = 1 to #iterations:

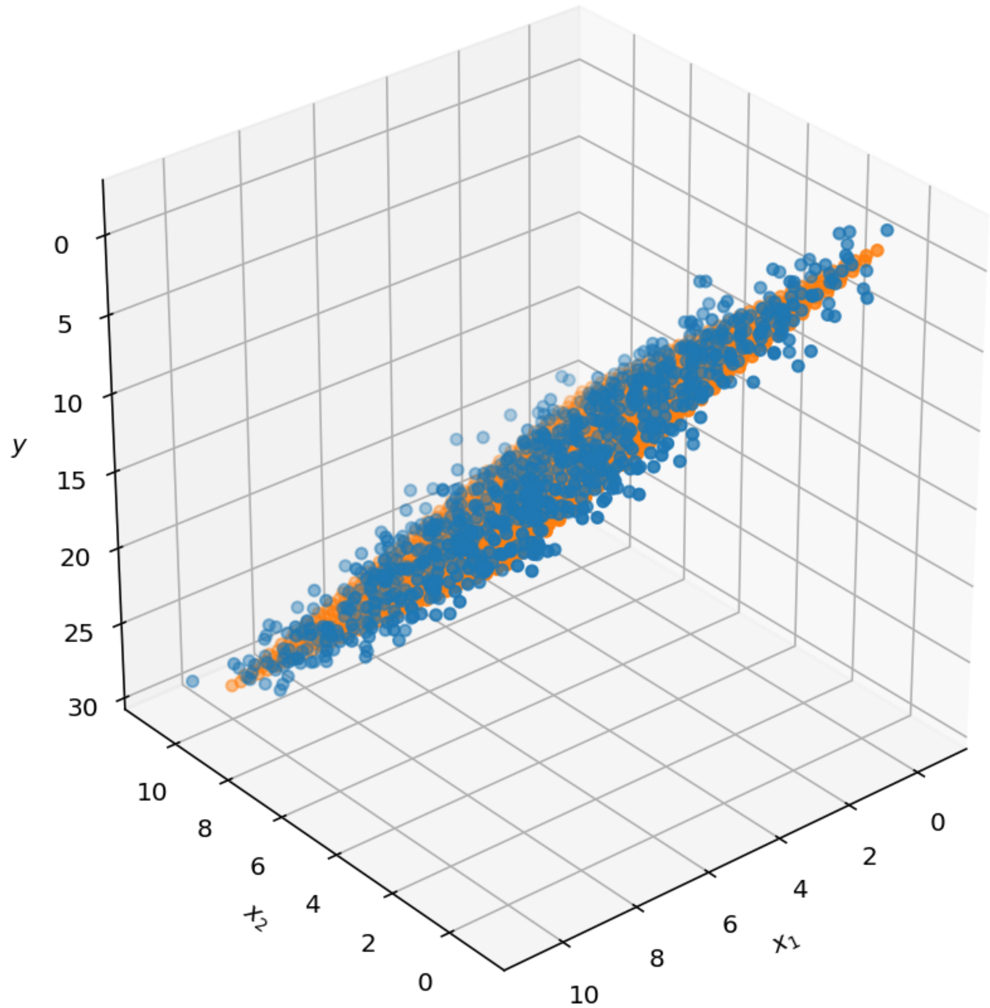
- **Step 2**. Compute the gradient $\nabla \mathcal{L}(W) = \frac{\partial \mathcal{L}(W)}{\partial W}$.
- **Step 3**. Update the parameters $W \leftarrow \mathcal{L}(W) = W - \eta \frac{\partial \mathcal{L}(W)}{\partial W}$ where η is the learning rate.

[6]:
```python
# Gradient of L(W) with respect to W.
def grad_L_W(X, Y, W):
    return 2*(torch.mm(X.T, torch.mm(X,W)-Y))
```

[7]:
```python
# y=w0+w1*x1+w2*x2
# Some settings.
print(X.shape, Y.shape)    # torch.Size([900, 3]) torch.Size([900, 1])
iterations    = 20000
learning_rate = 0.000001

# Gradient descent algorithm.
# Step 1. Initialize the parameters W.
W = torch.zeros(3,1)

for i in range(iterations):
    # Step 2. Calculate the gradient of L(W) w.r.t. W.
    grad = grad_L_W(X, Y, W)
    # Step 3. Update parameters W.
    W = W - learning_rate*grad####### To be filled #######  Hint: Use grad, W,␣
 ↪learning_rate.

#w0, w1, w2 = np.array(W).reshape(-1)
w0, w1, w2 = W[0,0].numpy(), W[1,0].numpy(), W[2,0].numpy()
print('y = {:.2f} + {:.2f}*x1 + {:.2f}*x2'.format(w0, w1, w2))
```

```
torch.Size([900, 3]) torch.Size([900, 1])
y = -0.69 + 0.98*x1 + 1.94*x2
```

[8]:
```python
# Visualization.
vis(w0, w1, w2)
```



0
5
10
15
20
25
30
y
10
8
6
4
2
0
x_2
0
2
4
6
8
10
x_1


```
[ ]:
```

Problem_2

May 1, 2020

[1]:
```
# -*- coding: utf-8 -*-
"""
Training an image classifier
----------------------------
We will do the following steps in order:
1. Load and normalizing the MNIST training and test datasets using
   ``torchvision``
2. Define a nearest neighbor classifier
3. Test the model on the test data (There is no training step for nearest␣
 ↪neighbor classifier).
1. Loading and normalizing MNIST
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Using ``torchvision``, it's extremely easy to load MNIST.
"""
import os
import torch
import torchvision
import torchvision.transforms as transforms

import itertools

########################################################################
# The output of torchvision datasets are PILImage images of range [0, 1].
# We transform them to Tensors of normalized range [-1, 1].
# .. note::
#     If running on Windows and you get a BrokenPipeError, try setting
#     the num_worker of torch.utils.data.DataLoader() to 0.
```

[2]:
```
pytorch_device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[3]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.1307,), (0.3081,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=60000,
```

```
                                                shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=16,
                                          shuffle=False, num_workers=2)

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

[4]:

```
########################################################################
# Let us show some of the training images, for fun.

import matplotlib.pyplot as plt
import numpy as np

# functions to show an image

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.cpu().numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap='gray')
    plt.show()

# get some random training images
examples = enumerate(trainloader)
batch_idx, (example_data, example_targets) = next(examples)

# show images
imshow(torchvision.utils.make_grid(example_data[:4]))

# print labels
print(' '.join('%5s' % classes[example_targets[j]] for j in range(4)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

<Figure size 640x480 with 1 Axes>

```
    1     1     6     2
```

[24]:

```
########################################################################
# 2. Define a nearest neighbor model
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
```

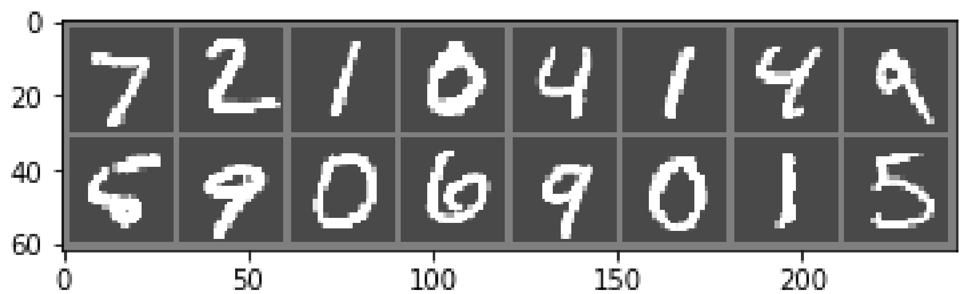
```
    def __init__(self):
        super(Model, self).__init__()
        self.database_x = example_data.reshape(60000, 28*28).to(pytorch_device)
        self.database_y = example_targets.to(pytorch_device)

    def forward(self, x):
        # shape of input (=x): [16, 1, 28, 28]
        # shape of output: [16]
        # output can take on integers in [0, 9]

        x = x.view(-1, 1 * 28 * 28)
        dists = -2*torch.mm(x, self.database_x.T) + torch.sum(self.
 ↪database_x**2, axis=1) \
                    + torch.sum(x**2, axis=1).unsqueeze(1)
#         print(dists.shape)

        prediction = []
        for i in range(x.shape[0]):
            idx = torch.topk(dists[i,:], k=1, largest=False)
            prediction.append(self.database_y[idx.indices])


        return prediction

model = Model().to(pytorch_device)
```

[ ]:
```
# xt = 60k x 728
# x = 16 x 728

dists = -2 * np.dot(X, self.X_train.T) + np.sum(self.X_train**2,    axis=1) +␣
 ↪np.sum(X**2, axis=1)[:, np.newaxis]
    return dists
```

[25]:
```
######################################################################
# See `here <https://pytorch.org/docs/stable/notes/serialization.html>`_
# for more details on saving PyTorch models.
#
# 5. Test the nearest neighbor model on the test data
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#
#
# We will check this by predicting the class label that the nearest neighbor␣
 ↪model
# outputs, and checking it against the ground-truth. If the prediction is
# correct, we add the sample to the list of correct predictions.
#
# Okay, first step. Let us display an image from the test set to get familiar.
```

```
dataiter = iter(testloader)
images, labels = dataiter.next()
images, labels = images.to(pytorch_device), labels.to(pytorch_device)

# print images
imshow(torchvision.utils.make_grid(images))

print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in␣
 ↪range(len(labels))))

########################################################################
# Okay, now let us see what the nearest neighbor model thinks these examples␣
 ↪above are:
outputs = model(images)

predicted = outputs
print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(len(labels))))
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



```
GroundTruth:      7     2     1     0     4     1     4     9     5     9     0
6     9     0     1     5
Predicted:      7     2     1     0     4     1     4     9     5     9     0
6     9     0     1     5
```

[48]:
```
########################################################################
# The results seem pretty good.
#
# Let us look at how the model performs on the whole dataset.

def eval(model):
    correct = 0
```

```
    total = 0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    cmt = torch.zeros(10,10, dtype=torch.int64)

    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(pytorch_device), labels.
 ↪to(pytorch_device)
            outputs = model(images)
            predicted = outputs
            total += labels.size(0)
            correct += (torch.Tensor(predicted).to(pytorch_device) == labels).
 ↪sum().item()

            c = (torch.Tensor(predicted).to(pytorch_device) == labels).squeeze()
            for i in range(len(labels)):
                label = labels[i]
                cmt[labels[i], predicted[i]] += 1
                class_correct[label] += c[i].item()
                class_total[label] += 1

    print('Accuracy of the model on the 10000 test images: %d %%' % (
        100 * correct / total))

    for i in range(10):
        print('Accuracy of %5s : %2d %%' % (
            classes[i], 100 * class_correct[i] / class_total[i]))
    return cmt

cmt = eval(model)
```

```
Accuracy of the model on the 10000 test images: 96 %
Accuracy of     0 : 99 %
Accuracy of     1 : 99 %
Accuracy of     2 : 96 %
Accuracy of     3 : 96 %
Accuracy of     4 : 96 %
Accuracy of     5 : 96 %
Accuracy of     6 : 98 %
Accuracy of     7 : 96 %
Accuracy of     8 : 94 %
Accuracy of     9 : 95 %
```

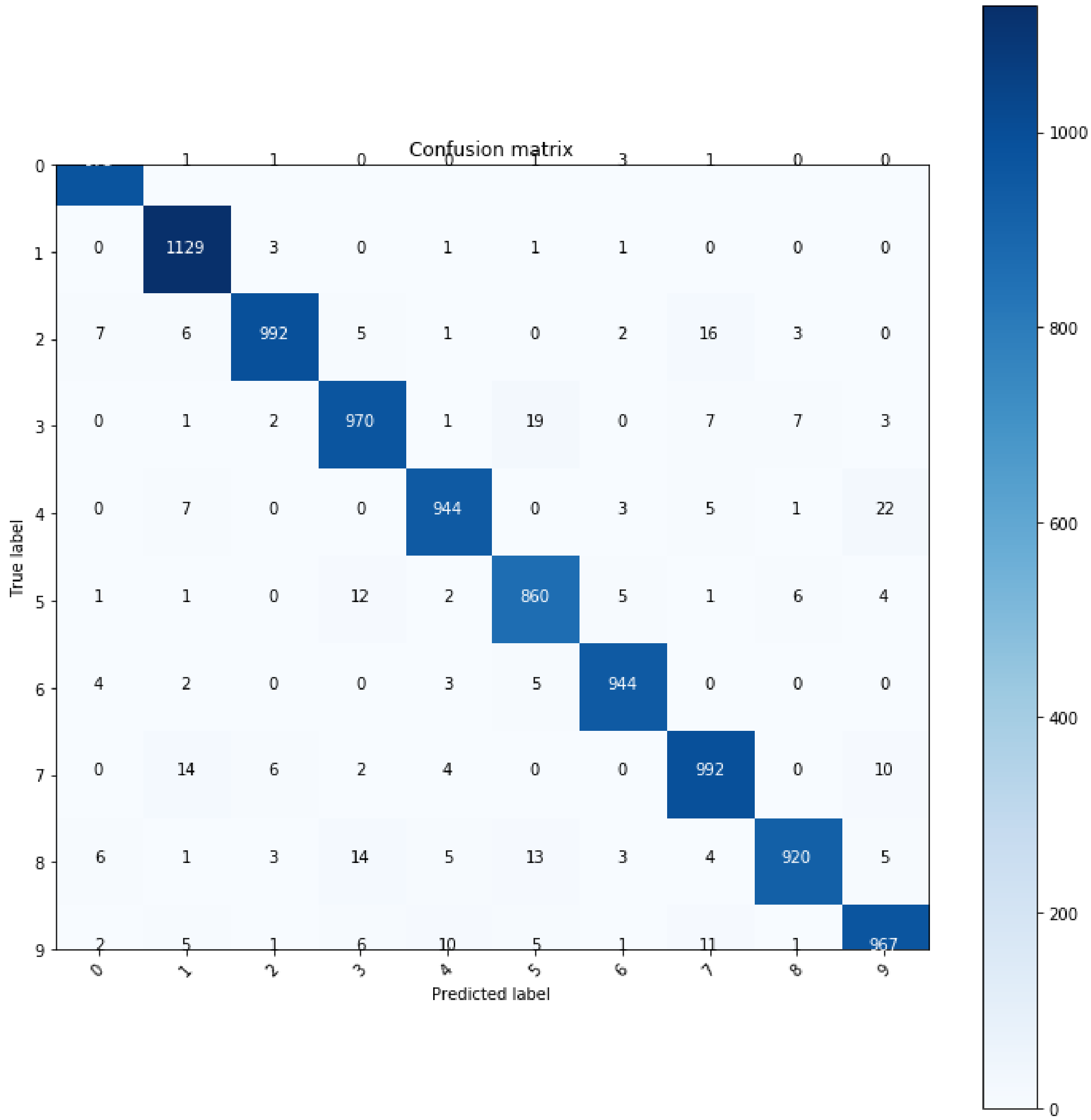
[28]:

```
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion␣
 ↪matrix', cmap=plt.cm.Blues):
```

```
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center",␣
 ↪color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[29]:
```
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cmt.numpy(), classes)
```

```
Confusion matrix, without normalization
[[ 973    1    1    0    0    1    3    1    0    0]
 [   0 1129    3    0    1    1    1    0    0    0]
 [   7    6  992    5    1    0    2   16    3    0]
 [   0    1    2  970    1   19    0    7    7    3]
 [   0    7    0    0  944    0    3    5    1   22]
 [   1    1    0   12    2  860    5    1    6    4]
 [   4    2    0    0    3    5  944    0    0    0]
 [   0   14    6    2    4    0    0  992    0   10]
 [   6    1    3   14    5   13    3    4  920    5]
 [   2    5    1    6   10    5    1   11    1  967]]
```



0.1 Experiments

0.2 Experiment with K - the number of neighbors

[85]:
```
class Model(nn.Module):
    def __init__(self, k=1):
        super(Model, self).__init__()
        self.database_x = example_data.reshape(60000, 28*28).to(pytorch_device)
        self.database_y = example_targets.to(pytorch_device)
        self.k = k
```

```
    def forward(self, x):
        # shape of input (=x): [16, 1, 28, 28]
        # shape of output: [16]
        # output can take on integers in [0, 9]

        x = x.view(-1, 1 * 28 * 28)
        dists = -2*torch.mm(x, self.database_x.T) + torch.sum(self.
↪database_x**2, axis=1) \
                    + torch.sum(x**2, axis=1).unsqueeze(1)
#         print(dists.shape)

        prediction = []
        for i in range(x.shape[0]):
            idx = torch.topk(dists[i,:], k=self.k, largest=False)
            y = self.database_y[idx.indices]
            y_unique = y.unique(sorted=True)
            y_unique_count = torch.stack([(y==y_u).sum() for y_u in y_unique])

            # print(y_unique, y_unique_count)
            # vote for the majority, the one with the most count
            prediction.append(y_unique[y_unique_count.argmax()])

        return prediction

model = Model(k=3).to(pytorch_device)
```

[86]:
```
model.k
```

[86]: 3

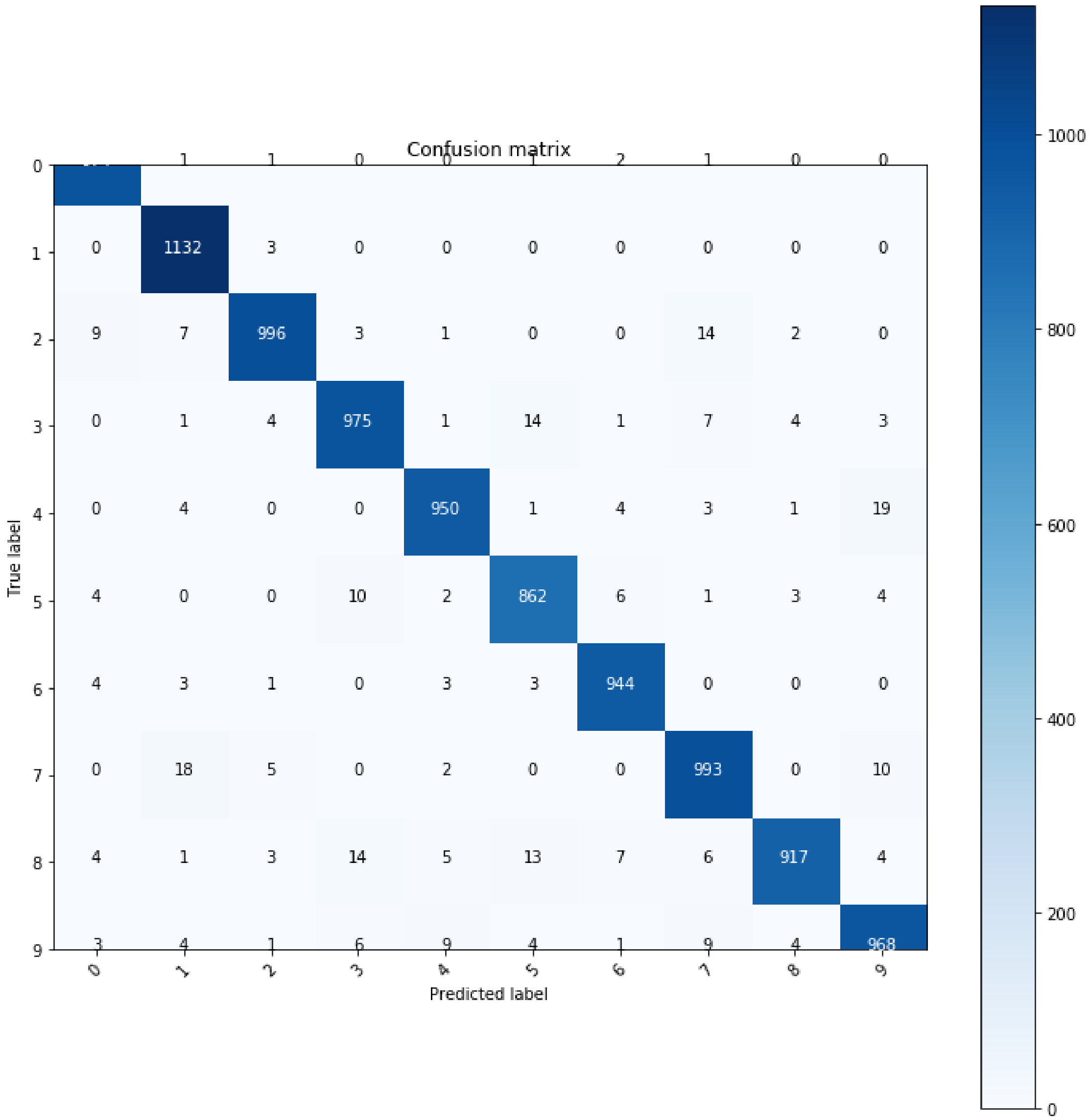
[87]:
```
cmt = eval(model)
```

```
Accuracy of the model on the 10000 test images: 97 %
Accuracy of     0 : 99 %
Accuracy of     1 : 99 %
Accuracy of     2 : 96 %
Accuracy of     3 : 96 %
Accuracy of     4 : 96 %
Accuracy of     5 : 96 %
Accuracy of     6 : 98 %
Accuracy of     7 : 96 %
Accuracy of     8 : 94 %
Accuracy of     9 : 95 %
```

1 accuracy increased using k=3 neighbors

[88]:
```
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cmt.numpy(), classes)
```

```
Confusion matrix, without normalization
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1132    3    0    0    0    0    0    0    0]
 [   9    7  996    3    1    0    0   14    2    0]
 [   0    1    4  975    1   14    1    7    4    3]
 [   0    4    0    0  950    1    4    3    1   19]
 [   4    0    0   10    2  862    6    1    3    4]
 [   4    3    1    0    3    3  944    0    0    0]
 [   0   18    5    0    2    0    0  993    0   10]
 [   4    1    3   14    5   13    7    6  917    4]
 [   3    4    1    6    9    4    1    9    4  968]]
```



1.0.1 misc code

```
[6]: # x = torch.randn(2, 5)
print(x)
idx = torch.topk(x[0,:], k=3)
print(idx.indices)

y = torch.from_numpy(np.array(range(10,1,-1)))
# y.gather(2,idx)
y[idx.indices]
```

```
tensor([[-0.4667, -0.3437,  1.4808, -1.4732, -0.7809],
        [ 0.2646, -0.7766,  1.1523, -2.0510, -0.1605]])
tensor([2, 1, 0])
```

[6]:
```
tensor([ 8,  9, 10])
```

[13]:
```python
x = torch.tensor([1, 2, 2, 3, 3])
x_unique = x.unique(sorted=True)
x_unique_count = torch.stack([(x==x_u).sum() for x_u in x_unique])
```

[14]:
```python
x_unique, x_unique_count
```

[14]:
```
(tensor([1, 2, 3]), tensor([1, 2, 2]))
```

[15]:
```python
x_unique_count.argmax(), x_unique[x_unique_count.argmax()]
```

[15]:
```
(tensor(2), tensor(3))
```

Problem_3

May 1, 2020

[1]:
```
# -*- coding: utf-8 -*-
"""
Training an image classifier
----------------------------
We will do the following steps in order:
1. Load and normalizing the MNIST training and test datasets using
   ``torchvision``
2. Define a SVM model
3. Define a loss function
4. Train the model on the training data
5. Test the model on the test data
1. Loading and normalizing MNIST
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Using ``torchvision``, it's extremely easy to load MNIST.
"""
import torch
import torchvision
import torchvision.transforms as transforms
import itertools

########################################################################
# The output of torchvision datasets are PILImage images of range [0, 1].
# We transform them to Tensors of normalized range [-1, 1].
# .. note::
#     If running on Windows and you get a BrokenPipeError, try setting
#     the num_worker of torch.utils.data.DataLoader() to 0.
```

[2]:
```
pytorch_device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[28]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.1307,), (0.3081,))])

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=16,
                                          shuffle=True, num_workers=2)
```

```
testset = torchvision.datasets.MNIST(root='./data', train=False,
                                      download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=16,
                                         shuffle=False, num_workers=2)

classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

[4]:
```
########################################################################
# Let us show some of the training images, for fun.

import matplotlib.pyplot as plt
import numpy as np

# functions to show an image

def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.cpu().numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)), cmap='gray')
    plt.show()

# get some random training images
examples = enumerate(trainloader)
batch_idx, (example_data, example_targets) = next(examples)

# show images
imshow(torchvision.utils.make_grid(example_data[:4]))


# print labels
print(' '.join('%5s' % classes[example_targets[j]] for j in range(4)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
<Figure size 640x480 with 1 Axes>
```

```
    5     7     0     0
```

[5]:
```
########################################################################
# 2. Define a logistic regression model
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

import torch.nn as nn
import torch.nn.functional as F
```

```
class Model(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(Model, self).__init__()
        self.layer = nn.Linear(in_dim, out_dim)

    def forward(self, x):
        # shape of input (=x): [16, 1, 28, 28]
        # shape of output: [16, 10]
        x = x.view(-1, 1 * 28 * 28)
        prediction = self.layer(x)
        return prediction

model = Model(28*28, 10).to(pytorch_device)
```

[30]:

```
########################################################################
# 3. Define a Loss function and optimizer
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Let's use a Classification Cross-Entropy loss and SGD with momentum.

import torch.optim as optim

learning_rate = 0.01
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

[31]:

```
########################################################################
# 4. Train the model
# ^^^^^^^^^^^^^^^^^^^^
#
# This is when things start to get interesting.
# We simply have to loop over our data iterator, and feed the inputs to the
# model and optimize.

def train(model, epochs=2, printStep=2000):
    for epoch in range(epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data
            inputs, labels = inputs.to(pytorch_device), labels.
 ↪to(pytorch_device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
```

```
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % printStep == printStep-1:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / printStep))
                running_loss = 0.0

    print('Finished Training')

train(model)
```

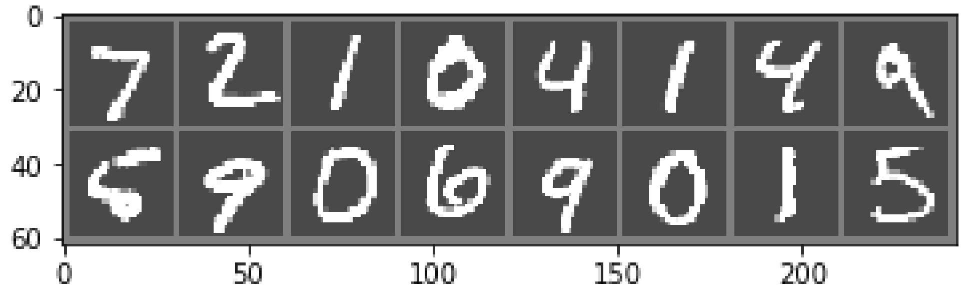
```
[1,  2000] loss: 0.278
[2,  2000] loss: 0.271
Finished Training
```

[22]:

```
########################################################################
# See `here <https://pytorch.org/docs/stable/notes/serialization.html>`_
# for more details on saving PyTorch models.
#
# 5. Test the model on the test data
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#
# We have trained the model for 2 passes over the training dataset.
# But we need to check if the model has learnt anything at all.
#
# We will check this by predicting the class label that the model
# outputs, and checking it against the ground-truth. If the prediction is
# correct, we add the sample to the list of correct predictions.
#
# Okay, first step. Let us display an image from the test set to get familiar.

dataiter = iter(testloader)
images, labels = dataiter.next()
images, labels = images.to(pytorch_device), labels.to(pytorch_device)

# print images
imshow(torchvision.utils.make_grid(images))

print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in␣
 ↪range(len(labels))))

########################################################################
```

```
# Okay, now let us see what the model thinks these examples above are:

outputs = model(images)

########################################################################
# The outputs are energies for the 10 classes.
# The higher the energy for a class, the more the model
# thinks that the image is of the particular class.
# So, let's get the index of the highest energy:
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(len(labels))))
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).



GroundTruth: 7   2   1   0   4   1   4   9   5   9   0
6   9   0   1   5
Predicted: 7   2   1   0   4   1   4   9   6   9   0
6   9   0   1   5

[23]:

```
########################################################################
# The results seem pretty good.
#
# Let us look at how the model performs on the whole dataset.

def eval(model):
    correct = 0
    total = 0

    with torch.no_grad():
        for data in trainloader:
            images, labels = data
```

```
            images, labels = images.to(pytorch_device), labels.
↪to(pytorch_device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the 60000 train images: %d %%' % (
        100 * correct / total))

    correct = 0
    total = 0

    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    cmt = torch.zeros(10,10, dtype=torch.int64)


    with torch.no_grad():
        for data in testloader:
            images, labels = data
            images, labels = images.to(pytorch_device), labels.
↪to(pytorch_device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            c = (predicted == labels).squeeze()
            for i in range(len(labels)):
                label = labels[i]
                cmt[labels[i], predicted[i]] += 1
                class_correct[label] += c[i].item()
                class_total[label] += 1


    print('Accuracy of the model on the 10000 test images: %d %%' % (
        100 * correct / total))

    for i in range(10):
        print('Accuracy of %5s : %2d %%' % (
            classes[i], 100 * class_correct[i] / class_total[i]))
    return cmt

cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
```

```
Accuracy of the model on the 10000 test images: 92 %
Accuracy of     0 : 97 %
Accuracy of     1 : 97 %
Accuracy of     2 : 88 %
Accuracy of     3 : 91 %
Accuracy of     4 : 93 %
Accuracy of     5 : 83 %
Accuracy of     6 : 95 %
Accuracy of     7 : 91 %
Accuracy of     8 : 91 %
Accuracy of     9 : 89 %
```

[10]:

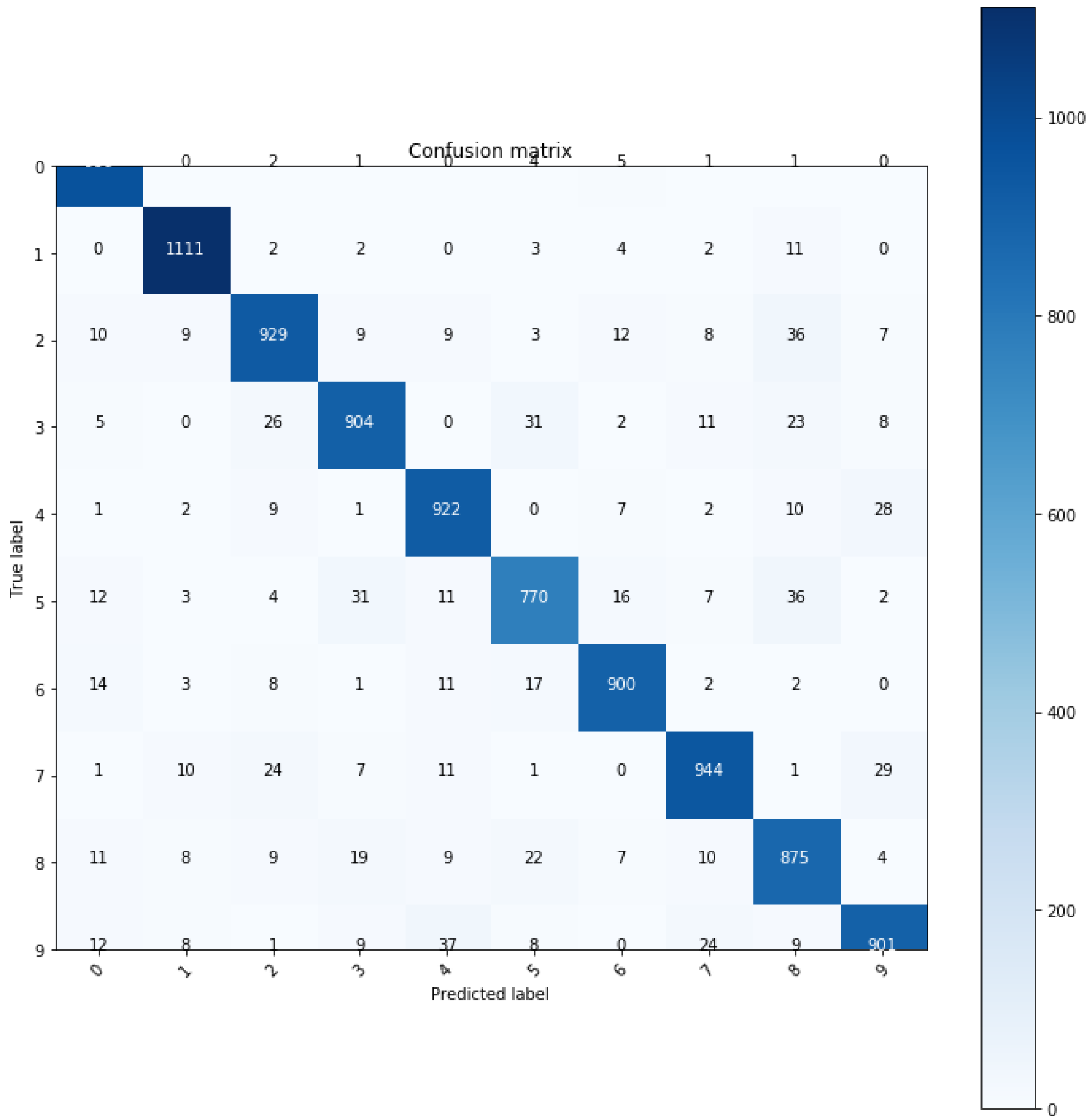
```python
def plot_confusion_matrix(cm, classes, normalize=False, title='Confusion␣
↪matrix', cmap=plt.cm.Blues):
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center",␣
↪color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[14]:

```python
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cmt.numpy(), classes)
```

```
Confusion matrix, without normalization
[[ 966    0    2    1    0    4    5    1    1    0]
 [   0 1111    2    2    0    3    4    2   11    0]
 [  10    9  929    9    9    3   12    8   36    7]
 [   5    0   26  904    0   31    2   11   23    8]
 [   1    2    9    1  922    0    7    2   10   28]
 [  12    3    4   31   11  770   16    7   36    2]
```

```
[  14    3    8    1   11   17  900    2    2    0]
[   1   10   24    7   11    1    0  944    1   29]
[  11    8    9   19    9   22    7   10  875    4]
[  12    8    1    9   37    8    0   24    9  901]]
```



0.1 Experiments

0.1.1 train for more epochs

[30]:
```
model = Model(28*28, 10).to(pytorch_device)

learning_rate = 0.01
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train(model, epochs=5)
```

```
[1,  2000] loss: 0.410
[2,  2000] loss: 0.301
[3,  2000] loss: 0.284
[4,  2000] loss: 0.279
[5,  2000] loss: 0.279
Finished Training
```

[31]:
```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 92 %
Accuracy of     0 : 98 %
Accuracy of     1 : 97 %
Accuracy of     2 : 89 %
Accuracy of     3 : 89 %
Accuracy of     4 : 93 %
Accuracy of     5 : 85 %
Accuracy of     6 : 95 %
Accuracy of     7 : 94 %
Accuracy of     8 : 88 %
Accuracy of     9 : 88 %
```

[32]:
```
# train for 5 more epochs
train(model, epochs=5)
```

```
[1,  2000] loss: 0.275
[2,  2000] loss: 0.263
[3,  2000] loss: 0.268
[4,  2000] loss: 0.264
[5,  2000] loss: 0.259
Finished Training
```

[33]:
```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 91 %
```

```
Accuracy of     0 : 97 %
Accuracy of     1 : 98 %
Accuracy of     2 : 90 %
Accuracy of     3 : 86 %
Accuracy of     4 : 92 %
Accuracy of     5 : 91 %
Accuracy of     6 : 92 %
Accuracy of     7 : 92 %
Accuracy of     8 : 85 %
Accuracy of     9 : 91 %
```

model started over-fitting maybe because of higher learning rate

0.1.2 change learning rate

[36]:
```
model = Model(28*28, 10).to(pytorch_device)

learning_rate = 0.001
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train(model, epochs=10)
```

```
[1,  2000] loss: 0.740
[2,  2000] loss: 0.398
[3,  2000] loss: 0.353
[4,  2000] loss: 0.334
[5,  2000] loss: 0.326
[6,  2000] loss: 0.313
[7,  2000] loss: 0.311
[8,  2000] loss: 0.304
[9,  2000] loss: 0.298
[10,  2000] loss: 0.295
Finished Training
```

[37]:
```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 91 %
Accuracy of the model on the 10000 test images: 91 %
Accuracy of     0 : 97 %
Accuracy of     1 : 97 %
Accuracy of     2 : 89 %
Accuracy of     3 : 90 %
Accuracy of     4 : 92 %
Accuracy of     5 : 86 %
Accuracy of     6 : 94 %
Accuracy of     7 : 90 %
```

```
Accuracy of     8 : 89 %
Accuracy of     9 : 89 %
```

[38]:
```
train(model, epochs=10)
```

```
[1,  2000] loss: 0.290
[2,  2000] loss: 0.291
[3,  2000] loss: 0.284
[4,  2000] loss: 0.290
[5,  2000] loss: 0.288
[6,  2000] loss: 0.285
[7,  2000] loss: 0.285
[8,  2000] loss: 0.282
[9,  2000] loss: 0.280
[10,  2000] loss: 0.272
Finished Training
```

[39]:
```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 92 %
Accuracy of     0 : 97 %
Accuracy of     1 : 97 %
Accuracy of     2 : 88 %
Accuracy of     3 : 90 %
Accuracy of     4 : 93 %
Accuracy of     5 : 86 %
Accuracy of     6 : 94 %
Accuracy of     7 : 92 %
Accuracy of     8 : 89 %
Accuracy of     9 : 89 %
```

0.001 learning rate with more epochs is more stable and gives better optimum

0.2 Batch size

as the model is overfitting, lets try to reduce the batch size

[32]:
```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)
```

[33]:
```
model = Model(28*28, 10).to(pytorch_device)

learning_rate = 0.01
criterion = torch.nn.CrossEntropyLoss()
```

```
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train(model, epochs=10, printStep=200)
```

```
[1,   200] loss: 0.736
[1,   400] loss: 0.444
[1,   600] loss: 0.401
[1,   800] loss: 0.370
[2,   200] loss: 0.343
[2,   400] loss: 0.339
[2,   600] loss: 0.339
[2,   800] loss: 0.323
[3,   200] loss: 0.322
[3,   400] loss: 0.303
[3,   600] loss: 0.325
[3,   800] loss: 0.304
[4,   200] loss: 0.298
[4,   400] loss: 0.298
[4,   600] loss: 0.298
[4,   800] loss: 0.306
[5,   200] loss: 0.304
[5,   400] loss: 0.291
[5,   600] loss: 0.282
[5,   800] loss: 0.299
[6,   200] loss: 0.300
[6,   400] loss: 0.284
[6,   600] loss: 0.289
[6,   800] loss: 0.280
[7,   200] loss: 0.281
[7,   400] loss: 0.285
[7,   600] loss: 0.282
[7,   800] loss: 0.283
[8,   200] loss: 0.290
[8,   400] loss: 0.273
[8,   600] loss: 0.284
[8,   800] loss: 0.279
[9,   200] loss: 0.277
[9,   400] loss: 0.280
[9,   600] loss: 0.280
[9,   800] loss: 0.269
[10,   200] loss: 0.272
[10,   400] loss: 0.272
[10,   600] loss: 0.288
[10,   800] loss: 0.271
Finished Training
```

[34]:

```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 92 %
Accuracy of     0 : 98 %
Accuracy of     1 : 97 %
Accuracy of     2 : 89 %
Accuracy of     3 : 91 %
Accuracy of     4 : 92 %
Accuracy of     5 : 85 %
Accuracy of     6 : 95 %
Accuracy of     7 : 92 %
Accuracy of     8 : 88 %
Accuracy of     9 : 90 %
```

[36]:

```
learning_rate = 0.0001
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train(model, epochs=5, printStep=200)
```

```
[1,   200] loss: 0.257
[1,   400] loss: 0.277
[1,   600] loss: 0.269
[1,   800] loss: 0.269
[2,   200] loss: 0.276
[2,   400] loss: 0.269
[2,   600] loss: 0.264
[2,   800] loss: 0.259
[3,   200] loss: 0.275
[3,   400] loss: 0.265
[3,   600] loss: 0.278
[3,   800] loss: 0.263
[4,   200] loss: 0.264
[4,   400] loss: 0.261
[4,   600] loss: 0.261
[4,   800] loss: 0.277
[5,   200] loss: 0.279
[5,   400] loss: 0.257
[5,   600] loss: 0.270
[5,   800] loss: 0.266
Finished Training
```

[37]:

```
cmt = eval(model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 92 %
Accuracy of     0 : 98 %
Accuracy of     1 : 97 %
Accuracy of     2 : 89 %
Accuracy of     3 : 90 %
```

```
Accuracy of     4 : 93 %
Accuracy of     5 : 86 %
Accuracy of     6 : 95 %
Accuracy of     7 : 92 %
Accuracy of     8 : 88 %
Accuracy of     9 : 89 %
```

Tried different batch sizes, higher batch size can be coupled with higher learning learning rate as the gradient will be less erratic as it is an average. with lower batch size we need to use lower learning rate. In this case changing batch size did not effect the performance

1 SVM

[40]:
```
class SVM_Model(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(SVM_Model, self).__init__()
        self.layer = nn.Linear(in_dim, out_dim)

    def forward(self, x):
        # shape of input (=x): [16, 1, 28, 28]
        # shape of output: [16, 10]
        x = x.view(-1, 1 * 28 * 28)
        prediction = self.layer(x)
        return prediction
```

[46]:
```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=16,
                                          shuffle=True, num_workers=2)
```

[49]:
```
svm_model = SVM_Model(28*28, 10).to(pytorch_device)

learning_rate = 0.001
criterion = torch.nn.MultiMarginLoss()
optimizer = optim.SGD(svm_model.parameters(), lr=learning_rate)

train(svm_model, epochs=7, printStep=2000)
```

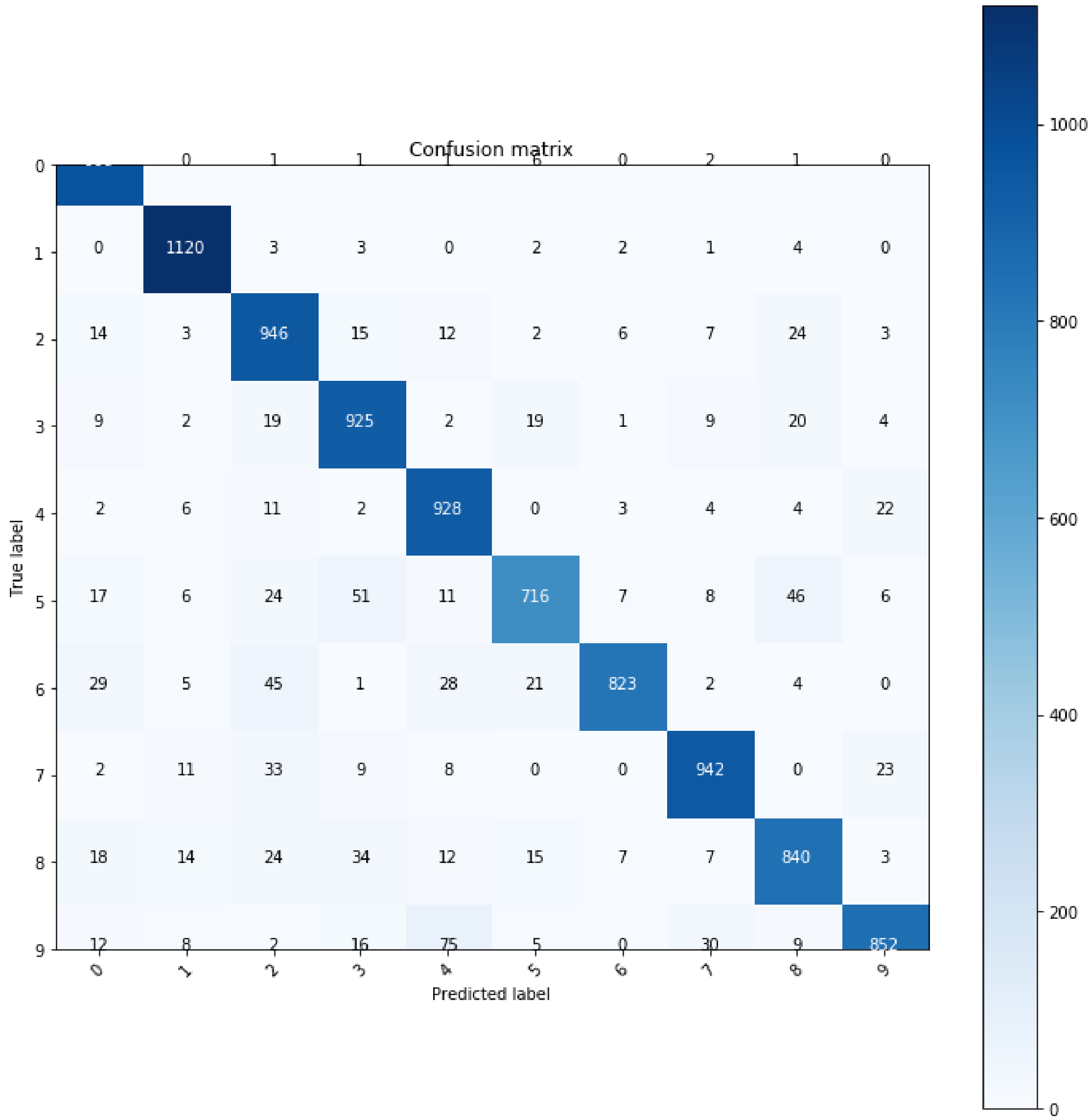
```
[1,  2000] loss: 0.147
[2,  2000] loss: 0.064
[3,  2000] loss: 0.055
[4,  2000] loss: 0.052
[5,  2000] loss: 0.048
[6,  2000] loss: 0.046
[7,  2000] loss: 0.046
Finished Training
```

[50]:
```
cmt = eval(svm_model)
```

```
Accuracy of the model on the 60000 train images: 90 %
Accuracy of the model on the 10000 test images: 91 %
Accuracy of     0 : 96 %
Accuracy of     1 : 96 %
Accuracy of     2 : 87 %
Accuracy of     3 : 89 %
Accuracy of     4 : 93 %
Accuracy of     5 : 85 %
Accuracy of     6 : 93 %
Accuracy of     7 : 91 %
Accuracy of     8 : 87 %
Accuracy of     9 : 88 %
```

[58]:
```
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cmt.numpy(), classes)
```

```
Confusion matrix, without normalization
[[ 968    0    1    1    1    6    0    2    1    0]
 [    0 1120    3    3    0    2    2    1    4    0]
 [   14    3  946   15   12    2    6    7   24    3]
 [    9    2   19  925    2   19    1    9   20    4]
 [    2    6   11    2  928    0    3    4    4   22]
 [   17    6   24   51   11  716    7    8   46    6]
 [   29    5   45    1   28   21  823    2    4    0]
 [    2   11   33    9    8    0    0  942    0   23]
 [   18   14   24   34   12   15    7    7  840    3]
 [   12    8    2   16   75    5    0   30    9  852]]
```



The test accuracy is higher than training accuracy here indicating that it is a better model than the logistic regression. There is no overfitting to training data.

[61]:
```
svm_model = SVM_Model(28*28, 10).to(pytorch_device)

learning_rate = 0.01
criterion = torch.nn.MultiMarginLoss()
optimizer = optim.SGD(svm_model.parameters(), lr=learning_rate)

train(svm_model, epochs=7, printStep=2000)
```

```
[1,  2000] loss: 0.069
[2,  2000] loss: 0.045
```

```
[3,  2000] loss: 0.041
[4,  2000] loss: 0.039
[5,  2000] loss: 0.037
[6,  2000] loss: 0.038
[7,  2000] loss: 0.037
Finished Training
```

[62]:
```
cmt = eval(svm_model)
```

```
Accuracy of the model on the 60000 train images: 92 %
Accuracy of the model on the 10000 test images: 91 %
Accuracy of     0 : 97 %
Accuracy of     1 : 97 %
Accuracy of     2 : 88 %
Accuracy of     3 : 91 %
Accuracy of     4 : 94 %
Accuracy of     5 : 85 %
Accuracy of     6 : 94 %
Accuracy of     7 : 91 %
Accuracy of     8 : 86 %
Accuracy of     9 : 90 %
```

Problem4

May 1, 2020

[1]:
```python
import cv2
import numpy as np
```

[53]:
```python
train_data = np.load('train_data.npy')
train_targets = np.load('train_targets.npy')

test_data = np.load('test_data.npy')
test_targets = np.load('test_targets.npy')
```

[30]:
```python
!pip install sklearn
```

```
WARNING: The directory '/home/abhilash/.cache/pip' or its parent directory
is not owned or is not writable by the current user. The cache has been
disabled. Check the permissions and owner of that directory. If executing pip
with sudo, you may want sudo's -H flag.
Collecting sklearn
  Downloading sklearn-0.0.tar.gz (1.1 kB)
Collecting scikit-learn
  Downloading scikit_learn-0.22.2.post1-cp36-cp36m-manylinux1_x86_64.whl (7.1
MB)
     |                 | 7.1 MB 2.4 MB/s eta 0:00:01
Collecting scipy>=0.17.0
  Downloading scipy-1.4.1-cp36-cp36m-manylinux1_x86_64.whl (26.1 MB)
     |                 | 26.1 MB 26.8 MB/s ta 0:00:01
Requirement already satisfied: numpy>=1.11.0 in
./imgrecog/lib/python3.6/site-packages (from scikit-learn->sklearn) (1.18.3)
Collecting joblib>=0.11
  Downloading joblib-0.14.1-py2.py3-none-any.whl (294 kB)
     |                 | 294 kB 15.2 MB/s eta 0:00:01
Building wheels for collected packages: sklearn
  Building wheel for sklearn (setup.py) … done
  Created wheel for sklearn: filename=sklearn-0.0-py2.py3-none-any.whl
size=1315
sha256=215df46fae84b8e20c23f74c31492a665d7c8f0bd1e392a687744561e3d0e9aa
  Stored in directory: /tmp/pip-ephem-wheel-cache-
jk1sotph/wheels/23/9d/42/5ec745cbbb17517000a53cecc49d6a865450d1f5cb16dc8a9c
```

```
Successfully built sklearn
Installing collected packages: scipy, joblib, scikit-learn, sklearn
Successfully installed joblib-0.14.1 scikit-learn-0.22.2.post1 scipy-1.4.1
sklearn-0.0
```

[17]:
```
from sklearn.cluster import KMeans
from sklearn import preprocessing
import copy
from tqdm.notebook import tqdm
```

[8]:
```
# compute dense SIFT
def computeSIFT(data):
    x = []
    for i in tqdm(range(0, len(data))):
        sift = cv2.xfeatures2d.SIFT_create()
#         orb = cv2.ORB_create(nfeatures=1500)
        img = data[i]
        img = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX).astype('uint8')
        step_size = 5
        kp = [cv2.KeyPoint(x, y, step_size) for x in range(0, img.shape[0],␣
 ↪step_size) for y in range(0, img.shape[1], step_size)]
        dense_feat = sift.compute(img, kp)
#         kp, dense_feat = orb.detectAndCompute(img, None)
        x.append(dense_feat[1])

    return x
```

[58]:
```
x_train = computeSIFT(train_data.squeeze())
x_test = computeSIFT(test_data.squeeze())
```

[59]:
```
all_train_desc = []
for i in range(len(x_train)):
    for j in range(x_train[i].shape[0]):
        all_train_desc.append(x_train[i][j,:])

all_train_desc = np.array(all_train_desc)
```

[4]:
```
from sklearn.neighbors import KNeighborsClassifier

# train model
def trainKNN(data, labels, k):
    neigh = KNeighborsClassifier(n_neighbors=k, p=2)
    neigh.fit(data, labels)
    return neigh
```

[5]:
```
# build BoW presentation from SIFT of training images
def clusterFeatures(all_train_desc, k):
```

```
    kmeans = KMeans(n_clusters=k, random_state=0).fit(all_train_desc)
    return kmeans
```

[6]:
```
# form training set histograms for each training image using BoW representation
def formTrainingSetHistogram(x_train, kmeans, k):
    train_hist = []
    for i in range(len(x_train)):
        data = copy.deepcopy(x_train[i])
        predict = kmeans.predict(data)
        train_hist.append(np.bincount(predict, minlength=k).reshape(1,-1).
 ↪ravel())

    return np.array(train_hist)


# build histograms for test set and predict
def predictKMeans(kmeans, scaler, x_test, train_hist, train_label, k):
    # form histograms for test set as test data
    test_hist = formTrainingSetHistogram(x_test, kmeans, k)

    # make testing histograms zero mean and unit variance
    test_hist = scaler.transform(test_hist)

    # Train model using KNN
    knn = trainKNN(train_hist, train_label, k)
    predict = knn.predict(test_hist)
    return np.array([predict], dtype=np.array([test_targets]).dtype)


def accuracy(predict_label, test_label):
    return np.mean(np.array(predict_label.tolist()[0]) == np.array(test_label))
```

[60]:
```
k = [10, 15, 20, 25, 30, 35, 40]
for i in range(len(k)):
    kmeans = clusterFeatures(all_train_desc, k[i])
    train_hist = formTrainingSetHistogram(x_train, kmeans, k[i])

    # preprocess training histograms
    scaler = preprocessing.StandardScaler().fit(train_hist)
    train_hist = scaler.transform(train_hist)

    predict = predictKMeans(kmeans, scaler, x_test, train_hist, train_targets,␣
 ↪k[i])
    res = accuracy(predict, test_targets)
    print("k =", k[i], ", Accuracy:", res*100, "%")
```

```
k = 10 , Accuracy: 30.5 %
```

```
k = 15 , Accuracy: 36.6 %
k = 20 , Accuracy: 45.7 %
k = 25 , Accuracy: 59.5 %
k = 30 , Accuracy: 59.699999999999996 %
k = 35 , Accuracy: 65.0 %
k = 40 , Accuracy: 60.0 %
```

1 Bag of SIFT Representation + one-vs-all SVMs

[15]:
```
from sklearn.svm import LinearSVC
```

[62]:
```
k = 35  # 60
kmeans = clusterFeatures(all_train_desc, k)

# form training and testing histograms
train_hist = formTrainingSetHistogram(x_train, kmeans, k)
test_hist = formTrainingSetHistogram(x_test, kmeans, k)
```

[63]:
```
# normalize histograms
scaler = preprocessing.StandardScaler().fit(train_hist)
train_hist = scaler.transform(train_hist)
test_hist = scaler.transform(test_hist)
```

[65]:
```
for c in np.arange(0.0001, 0.1, 0.00198):
    clf = LinearSVC(random_state=0, C=c)
    clf.fit(train_hist, train_targets)
    predict = clf.predict(test_hist)
    print ("C =", c, ",\t Accuracy:", np.mean(predict == test_targets)*100, "%")
```

```
C = 0.0001 ,     Accuracy: 55.1 %
C = 0.00208 ,    Accuracy: 64.2 %
C = 0.00406 ,    Accuracy: 65.7 %
C = 0.00604 ,    Accuracy: 67.0 %
C = 0.00802 ,    Accuracy: 68.0 %
C = 0.009999999999999998 ,       Accuracy: 68.10000000000001 %
C = 0.01198 ,    Accuracy: 68.10000000000001 %
C = 0.01396 ,    Accuracy: 68.0 %
C = 0.01594 ,    Accuracy: 68.0 %
C = 0.01792 ,    Accuracy: 68.0 %
C = 0.019899999999999998 ,       Accuracy: 68.0 %
C = 0.02188 ,    Accuracy: 68.30000000000001 %
C = 0.02386 ,    Accuracy: 68.5 %
C = 0.02584 ,    Accuracy: 68.60000000000001 %
C = 0.02782 ,    Accuracy: 68.60000000000001 %
C = 0.0298 ,     Accuracy: 68.4 %
C = 0.03178 ,    Accuracy: 68.5 %
```

```
C = 0.033760000000000005 ,        Accuracy: 68.5 %
C = 0.03574 ,     Accuracy: 68.5 %
C = 0.037720000000000004 ,        Accuracy: 68.60000000000001 %
C = 0.0397 ,      Accuracy: 68.8 %
C = 0.04168 ,     Accuracy: 69.0 %
C = 0.043660000000000004 ,        Accuracy: 69.19999999999999 %
C = 0.04564 ,     Accuracy: 69.19999999999999 %
C = 0.04762 ,     Accuracy: 69.3 %
C = 0.049600000000000005 ,        Accuracy: 69.39999999999999 %
C = 0.05158 ,     Accuracy: 69.39999999999999 %
C = 0.05356 ,     Accuracy: 69.3 %
C = 0.055540000000000006 ,        Accuracy: 69.1 %
C = 0.05752 ,     Accuracy: 69.1 %
C = 0.059500000000000004 ,        Accuracy: 69.0 %
C = 0.06148 ,     Accuracy: 69.1 %
C = 0.06346 ,     Accuracy: 69.1 %
C = 0.06544 ,     Accuracy: 69.1 %
C = 0.06742000000000001 ,         Accuracy: 69.1 %
C = 0.0694 ,      Accuracy: 69.1 %
C = 0.07138 ,     Accuracy: 69.1 %
C = 0.07336000000000001 ,         Accuracy: 69.1 %
C = 0.07534 ,     Accuracy: 69.3 %
C = 0.07732 ,     Accuracy: 69.3 %
C = 0.0793 ,      Accuracy: 69.19999999999999 %
C = 0.08128 ,     Accuracy: 69.3 %
C = 0.08326 ,     Accuracy: 69.19999999999999 %
C = 0.08524 ,     Accuracy: 69.3 %
C = 0.08722 ,     Accuracy: 69.3 %
C = 0.0892 ,      Accuracy: 69.3 %
C = 0.09118 ,     Accuracy: 69.3 %
C = 0.09316 ,     Accuracy: 69.3 %
C = 0.09514 ,     Accuracy: 69.39999999999999 %
C = 0.09712 ,     Accuracy: 69.39999999999999 %
C = 0.09910000000000001 ,         Accuracy: 69.39999999999999 %
```

1.1 Improve performance with Spatial Pyramid Matching

[7]:
```python
import math

def extract_denseSIFT(img):
    DSIFT_STEP_SIZE = 5
    sift = cv2.xfeatures2d.SIFT_create()
    disft_step_size = DSIFT_STEP_SIZE
    keypoints = [cv2.KeyPoint(x, y, disft_step_size)
            for y in range(0, img.shape[0], disft_step_size)
                for x in range(0, img.shape[1], disft_step_size)]
```

```
    descriptors = sift.compute(img, keypoints)[1]

    #keypoints, descriptors = sift.detectAndCompute(gray, None)
    return descriptors


# form histogram with Spatial Pyramid Matching upto level L with codebook␣
↪kmeans and k codewords
def getImageFeaturesSPM(L, img, kmeans, k):
    # print('getImageFeaturesSPM: ', img.shape)
    img = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX).astype('uint8')
    W = img.shape[1]
    H = img.shape[0]
    h = []
    for l in range(L+1):
        w_step = math.floor(W/(2**l))
        h_step = math.floor(H/(2**l))
        x, y = 0, 0
        for i in range(1,2**l + 1):
            x = 0
            for j in range(1, 2**l + 1):
                desc = extract_denseSIFT(img[y:y+h_step, x:x+w_step])
                #print("type:",desc is None, "x:",x,"y:",y, "desc_size:",desc␣
↪is None)
                predict = kmeans.predict(desc)
                histo = np.bincount(predict, minlength=k).reshape(1,-1).ravel()
                weight = 2**(l-L)
                h.append(weight*histo)
                x = x + w_step
            y = y + h_step

    hist = np.array(h).ravel()
    # normalize hist
    dev = np.std(hist)
    hist -= np.mean(hist)
    hist /= dev
    return hist


# get histogram representation for training/testing data
def getHistogramSPM(L, data, kmeans, k):
    x = []
    for i in tqdm(range(len(data))):
        hist = getImageFeaturesSPM(L, data[i], kmeans, k)
        x.append(hist)
    return np.array(x)
```

[67]:
```
k = 200
kmeans = clusterFeatures(all_train_desc, k)
```

[77]:
```
train_histo = getHistogramSPM(2, train_data.squeeze(), kmeans, k)
test_histo = getHistogramSPM(2, test_data.squeeze(), kmeans, k)
```

[79]:
```
# train SVM
for c in np.arange(0.000307, 0.001, 0.0000462):
    clf = LinearSVC(random_state=0, C=c)
    clf.fit(train_histo, train_targets)
    predict = clf.predict(test_histo)
    print ("C =", c, ",\t\t Accuracy:", np.mean(predict == test_targets)*100,␣
 ↪"%")
```

```
C = 0.000307 ,           Accuracy: 93.30000000000001 %
C = 0.00035319999999999997 ,             Accuracy: 93.60000000000001 %
C = 0.00039939999999999995 ,             Accuracy: 93.8 %
C = 0.00044559999999999993 ,             Accuracy: 94.1 %
C = 0.0004917999999999999 ,              Accuracy: 94.19999999999999 %
C = 0.0005379999999999998 ,              Accuracy: 94.3 %
C = 0.0005841999999999999 ,              Accuracy: 94.39999999999999 %
C = 0.0006303999999999999 ,              Accuracy: 94.5 %
C = 0.0006765999999999999 ,              Accuracy: 94.39999999999999 %
C = 0.0007227999999999998 ,              Accuracy: 94.3 %
C = 0.0007689999999999998 ,              Accuracy: 94.39999999999999 %
C = 0.0008151999999999999 ,              Accuracy: 94.5 %
C = 0.0008613999999999998 ,              Accuracy: 94.5 %
C = 0.0009075999999999997 ,              Accuracy: 94.5 %
C = 0.0009537999999999998 ,              Accuracy: 94.5 %
C = 0.0009999999999999998 ,              Accuracy: 94.5 %
```

1.1.1 decrease clusters

[87]:
```
k = 50
kmeans = clusterFeatures(all_train_desc, k)

train_histo = getHistogramSPM(2, train_data.squeeze(), kmeans, k)
test_histo = getHistogramSPM(2, test_data.squeeze(), kmeans, k)
```

```
100%|      | 1000/1000 [00:23<00:00, 42.95it/s]
100%|      | 1000/1000 [00:17<00:00, 57.07it/s]
```

[88]:
```
# train SVM
for c in np.arange(0.000307, 0.001, 0.0000462):
    clf = LinearSVC(random_state=0, C=c)
    clf.fit(train_histo, train_targets)
```

```
    predict = clf.predict(test_histo)
    print ("C =", c, ",\t\t Accuracy:", np.mean(predict == test_targets)*100,␣
 ↪"%")
```

```
C = 0.000307 ,           Accuracy: 90.10000000000001 %
C = 0.00035319999999999997 ,            Accuracy: 90.4 %
C = 0.00039939999999999995 ,            Accuracy: 90.8 %
C = 0.00044559999999999993 ,            Accuracy: 90.9 %
C = 0.0004917999999999999 ,             Accuracy: 90.9 %
C = 0.0005379999999999998 ,             Accuracy: 91.10000000000001 %
C = 0.0005841999999999999 ,             Accuracy: 91.2 %
C = 0.0006303999999999999 ,             Accuracy: 91.3 %
C = 0.0006765999999999999 ,             Accuracy: 91.5 %
C = 0.0007227999999999998 ,             Accuracy: 91.7 %
C = 0.0007689999999999998 ,             Accuracy: 91.8 %
C = 0.0008151999999999999 ,             Accuracy: 91.60000000000001 %
C = 0.0008613999999999998 ,             Accuracy: 91.60000000000001 %
C = 0.0009075999999999997 ,             Accuracy: 91.60000000000001 %
C = 0.0009537999999999998 ,             Accuracy: 91.60000000000001 %
C = 0.0009999999999999998 ,             Accuracy: 91.60000000000001 %
```

2 working with all the data

[9]:
```
train_data = np.load('train_data_all.npy')
train_targets = np.load('train_targets_all.npy')

test_data = np.load('test_data_all.npy')
test_targets = np.load('test_targets_all.npy')
```

[10]:
```
print(train_data.shape, train_targets.shape)
print(test_data.shape, test_targets.shape)
```

```
(60000, 1, 28, 28) (60000,)
(10000, 1, 28, 28) (10000,)
```

[11]:
```
x_train = computeSIFT(train_data.squeeze())
x_test = computeSIFT(test_data.squeeze())
```

```
100%|      | 60000/60000 [00:14<00:00, 4073.03it/s]
100%|      | 10000/10000 [00:02<00:00, 4109.23it/s]
```

[12]:
```
all_train_desc = []
for i in range(len(x_train)):
    for j in range(x_train[i].shape[0]):
        all_train_desc.append(x_train[i][j,:])
```

```
all_train_desc = np.array(all_train_desc)
```

[13]:
```
k = 50
kmeans = clusterFeatures(all_train_desc, k)

train_histo = getHistogramSPM(2, train_data.squeeze(), kmeans, k)
test_histo = getHistogramSPM(2, test_data.squeeze(), kmeans, k)
```

```
100%|          | 60000/60000 [14:46<00:00, 67.69it/s]
100%|          | 10000/10000 [02:28<00:00, 67.26it/s]
```

[16]:
```
# train SVM
for c in np.arange(0.000307, 0.001, 0.0000462):
    clf = LinearSVC(random_state=0, C=c)
    clf.fit(train_histo, train_targets)
    predict = clf.predict(test_histo)
    print ("C =", c, ",\t\t Accuracy:", np.mean(predict == test_targets)*100,␣
 ↪"%")
```

```
C = 0.000307 ,           Accuracy: 95.5 %
C = 0.00035319999999999997 ,             Accuracy: 95.6 %
C = 0.00039939999999999995 ,             Accuracy: 95.65 %
C = 0.00044559999999999993 ,             Accuracy: 95.66 %
C = 0.0004917999999999999 ,              Accuracy: 95.71 %
C = 0.0005379999999999998 ,              Accuracy: 95.75 %
C = 0.0005841999999999999 ,              Accuracy: 95.78 %
C = 0.0006303999999999999 ,              Accuracy: 95.82000000000001 %
C = 0.0006765999999999999 ,              Accuracy: 95.84 %
C = 0.0007227999999999998 ,              Accuracy: 95.84 %
C = 0.0007689999999999998 ,              Accuracy: 95.87 %
C = 0.0008151999999999999 ,              Accuracy: 95.91 %
C = 0.0008613999999999998 ,              Accuracy: 95.93 %
C = 0.0009075999999999997 ,              Accuracy: 95.95 %
C = 0.0009537999999999998 ,              Accuracy: 95.97 %
C = 0.0009999999999999998 ,              Accuracy: 95.97 %
```