

Homework #4

CSE 546: Machine Learning

Michael Ross

Expectation Maximization

1. [3 points] *Pandora* is a streaming music company like *Spotify* that was known to buck the collaborative filtering trend¹ and instead paid an army of employees to create feature vectors for each song by hand. Suppose you work at a Pandora clone and have feature vectors $x_1, \dots, x_n \in \mathbb{R}^d$ for all n songs in your database, and a particular user, for some subset $\mathcal{S} \subset \{1, \dots, n\}$, has listened to song $i \in \mathcal{S}$ exactly $Y_i \in \{1, 2, \dots\}$ times. You would like to make a playlist for this user so you assume Y_i is Poisson distributed with mean $\mathbb{E}[Y_i|x_i] =: \lambda_i = e^{w^T x_i}$ for some weight vector $w \in \mathbb{R}^d$ reflecting the user's preferences. That is,

$$p(Y_i = y|x_i, w) = \frac{\lambda_i^y}{y!} e^{-\lambda_i} = \frac{e^{y x_i^T w}}{y!} e^{-e^{w^T x_i}}.$$

The maximum likelihood estimator is $\hat{w} = \arg \max_w \prod_{i \in \mathcal{S}} p(y_i|x_i, w)$. The idea is that you would then construct a playlist out of the items $i \in \{1, \dots, n\}$ that maximize $x_i^T \hat{w}$.

- The estimate \hat{w} has no closed-form solution. Can the optimization problem be transformed into a *convex* optimization problem? If so, suggest a method of solving for \hat{w} given $\{(x_i, y_i)\}_{i \in \mathcal{S}}$. (Hint: how do you show a function $f(x)$ is convex?).
- You solve for the \hat{w} for this user and make a playlist for her. Weeks later you look at her listening history and observe that sometimes she listens to a particular set of songs and skips over others, and at some other point she listens to a different set of songs and skips over others. You have the epiphany that users are human beings whose preferences differ with their mood (e.g., music for workouts, studying, being sad, etc.). You decide she has k music moods and aim to make k playlists, one for each mood that could be modeled by a different weight vector w . The problem is that you don't know which observation $i \in \mathcal{S}$ is assigned to which mood. Describe (in math and words) how you would use the EM algorithm to make these k playlists.

Regression with Side Information

2. [8 points] In linear regression we have seen how penalizing the ℓ_2 -norm of the weights (Ridge) and ℓ_1 -norm of the weights (Lasso) affect the resulting solutions. Using different loss functions and regularizers to obtain different desired behaviors is very popular in machine learning. In this problem we will explore some of these ideas by using a general convex optimization solver CVXPY: <http://www.cvxpy.org/> to solve the optimization problems we define. Using these kinds of general solvers can be slower than a highly tuned custom solver you write yourself (e.g., accelerated gradient descent with a tuned stepsize) but they make it easy to swap out loss functions or regularizers. One of the benefits of convex optimization is that no matter which solver or method is used (coordinate descent, SGD, gradient descent, Newton's, etc.) they all converge to the same function value (unlike non-convex optimization in neural networks where the optimization method itself affects the resulting solution).

First let's generate some data. Let $n = 50$ and $f(x) = 10 \sum_{k=1}^4 \mathbf{1}\{x \geq \frac{k}{5}\}$, noting that $f(x)$ is non-decreasing in x (i.e., $f(x) \geq f(z)$ whenever $x \geq z$). For $i = 1, \dots, n$ let each $x_i = \frac{i-1}{n-1}$ and $y_i = \mathbf{1}\{i \neq 25\}(f(x_i) + \epsilon_i)$ where

¹Methods like matrix completion can leverage massive user-bases rating lots of items, but suffer from the "cold-start" problem: you recommend songs based on people's rating history, but to learn who would like a *new* song you need lots of people to listen to that song, but that requires you to suggest it and possibly degrade recommendation performance.

$\epsilon_i \sim \mathcal{N}(0, 1)$. The case where $i = 25$ represents an outlier in the data.

In the last homework we solved a problem of the form $\arg \min_{\alpha} \sum_{i=1}^n \ell(y_i - \sum_{j=1}^n k(x_i, x_j) \alpha_j)$ where $\ell(z) = \ell_{ls}(z) := z^2$ was the least squares loss. Least squares is the MLE for Gaussian noise, but is very sensitive to outliers. A more robust loss is the Huber loss:

$$\ell_{huber}(z) = \begin{cases} z^2 & \text{if } |z| \leq 1 \\ 2|z| - 1 & \text{otherwise} \end{cases}$$

which acts like least squares close to 0 but like the absolute value far from 0. Moreover, define a matrix $D \in \{-1, 0, 1\}^{(n-1) \times n}$

$$D_{i,j} = \begin{cases} -1 & \text{if } i = j \\ 1 & \text{if } i = j - 1 \\ 0 & \text{otherwise} \end{cases}$$

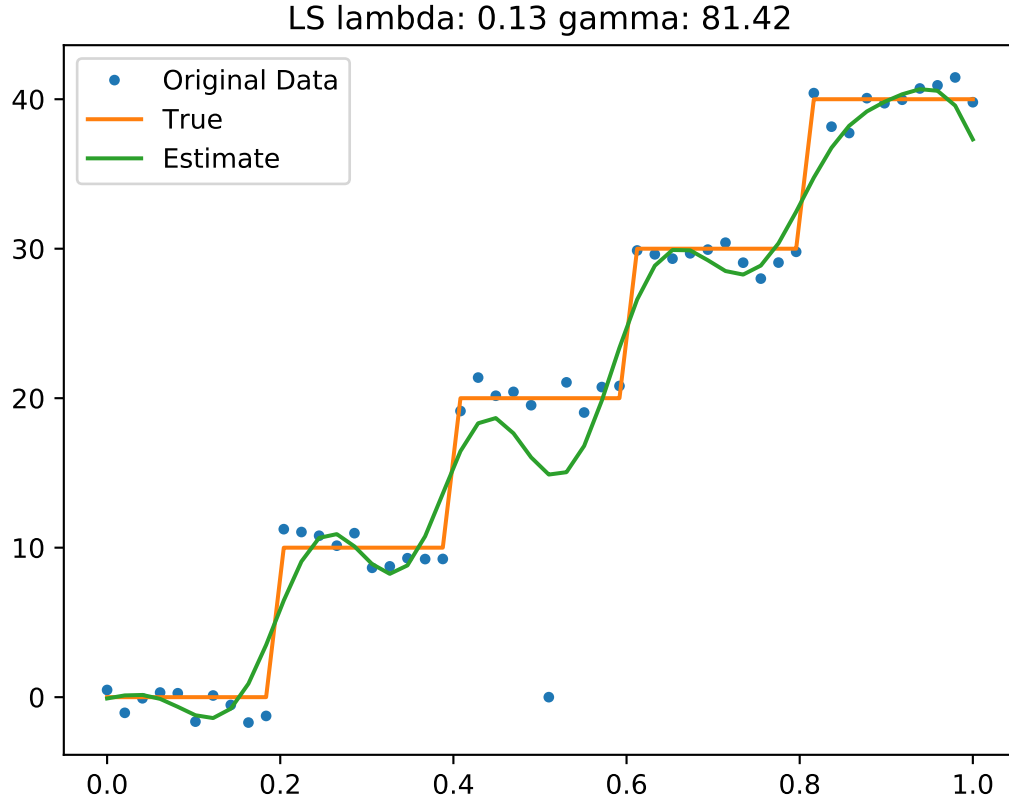
so that for any vector $z \in \mathbb{R}^n$ we have $Dz = (z_2 - z_1, z_3 - z_2, \dots, z_n - z_{n-1})$. In what follows let $k(x, z) = \exp(-\gamma \|x - z\|^2)$ where $\gamma > 0$ is a hyperparameter.

a. As a baseline, let

$$\hat{\alpha} = \arg \min_{\alpha} \sum_{i=1}^n \ell_{ls}(y_i - \sum_{j=1}^n K_{i,j} \alpha_j) + \lambda \alpha^T K \alpha, \quad \hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\hat{f}(x)$ found through leave-one-out CV. (Hint: start with the problem on the homepage of <http://www.cvxpy.org/> and modify it as needed.)

Answer:

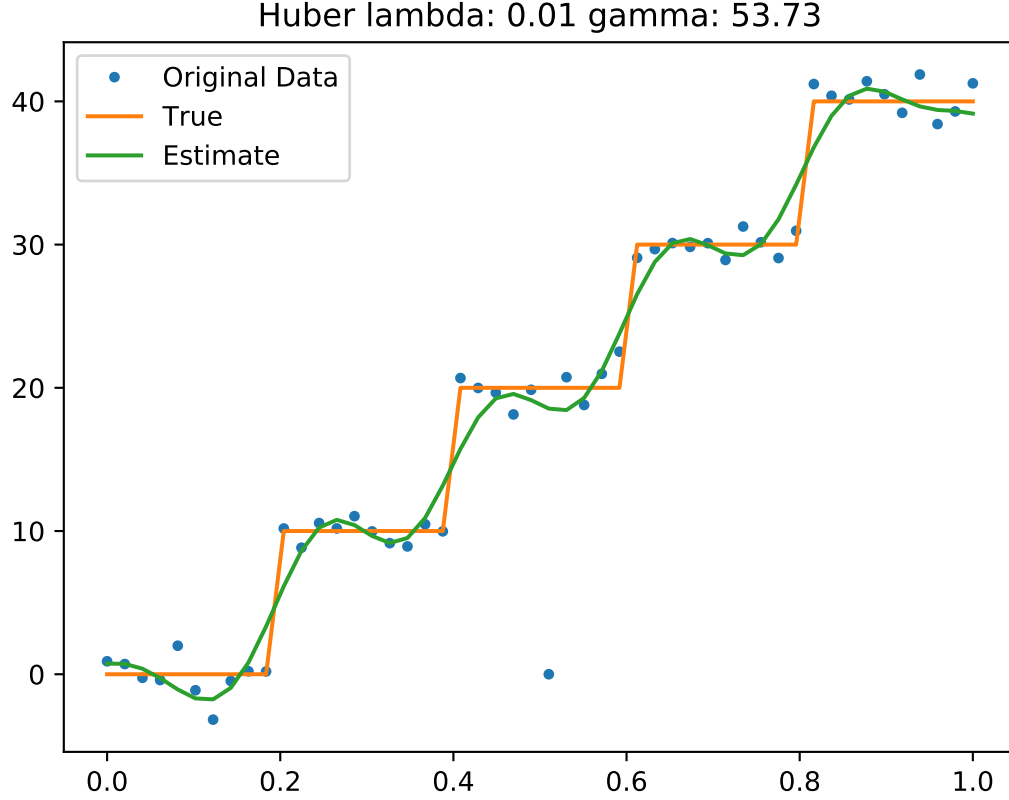


b. Now let

$$\hat{\alpha} = \arg \min_{\alpha} \sum_{i=1}^n \ell_{\text{huber}}(y_i - \sum_{j=1}^n K_{i,j} \alpha_j) + \lambda \alpha^T K \alpha, \quad \hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is the regularization constant. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\hat{f}(x)$ found through leave-one-out CV. (Hint: `huber` is a function in `cvxpy`.)

Answer:

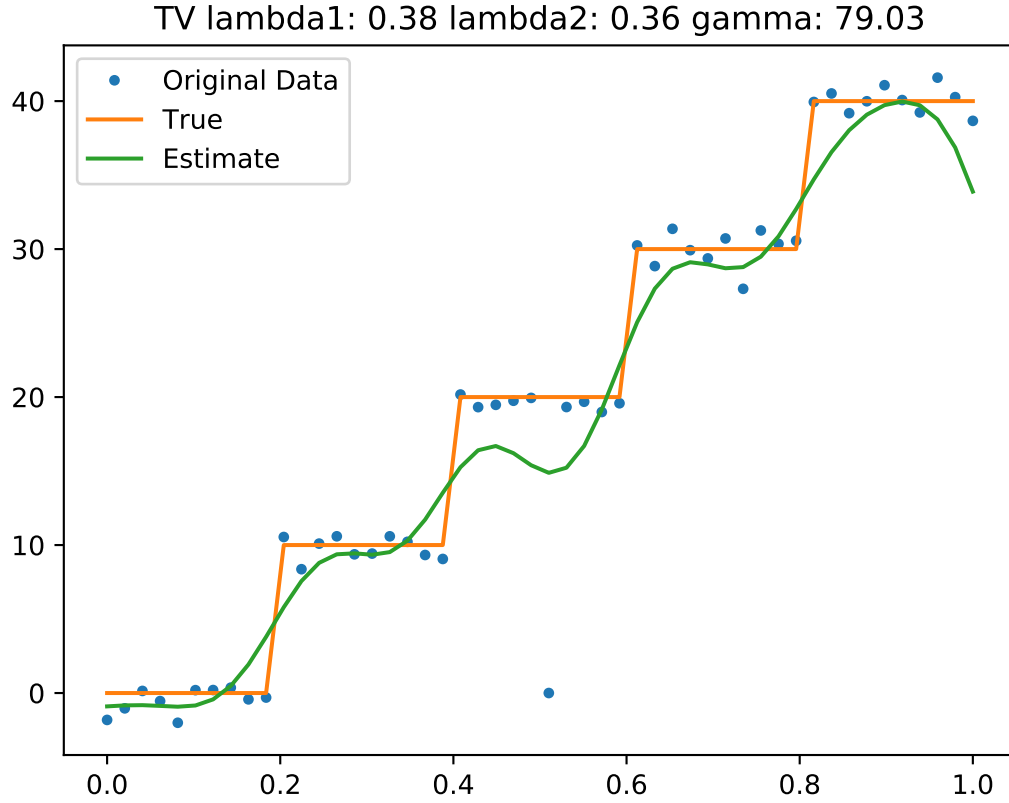


c. The total variation (TV) of a real-valued function g over $\{1, \dots, n\}$ is defined as $\sum_{i=1}^{n-1} |g_i - g_{i-1}| = \|Dg\|_1$ and is a common regularizer for de-noising a function (its two-dimensional counterpart is very popular for image de-noising or filling-in missing/damaged parts of photos). Let

$$\hat{\alpha} = \arg \min_{\alpha} \|K\alpha - y\|^2 + \lambda_1 \|DK\alpha\|_1 + \lambda_2 \alpha^T K \alpha, \quad \hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i k(x_i, x)$$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ_1, λ_2 are regularization constants. For intuition, the penalizer $\|DK\alpha\|_1$ prefers functions \hat{f} with sparse jumps in function value while $\alpha^T K \alpha$ prefers functions that are smoothly varying. On your own (not necessary to report plots), plot \hat{f} for a variety of values of $\gamma, \lambda_1, \lambda_2$ to see how they affect the solution. Use leave-one-out cross validation to find a good setting of $\gamma, \lambda_1, \lambda_2$. Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\hat{f}(x)$ found through leave-one-out CV.

Answer:



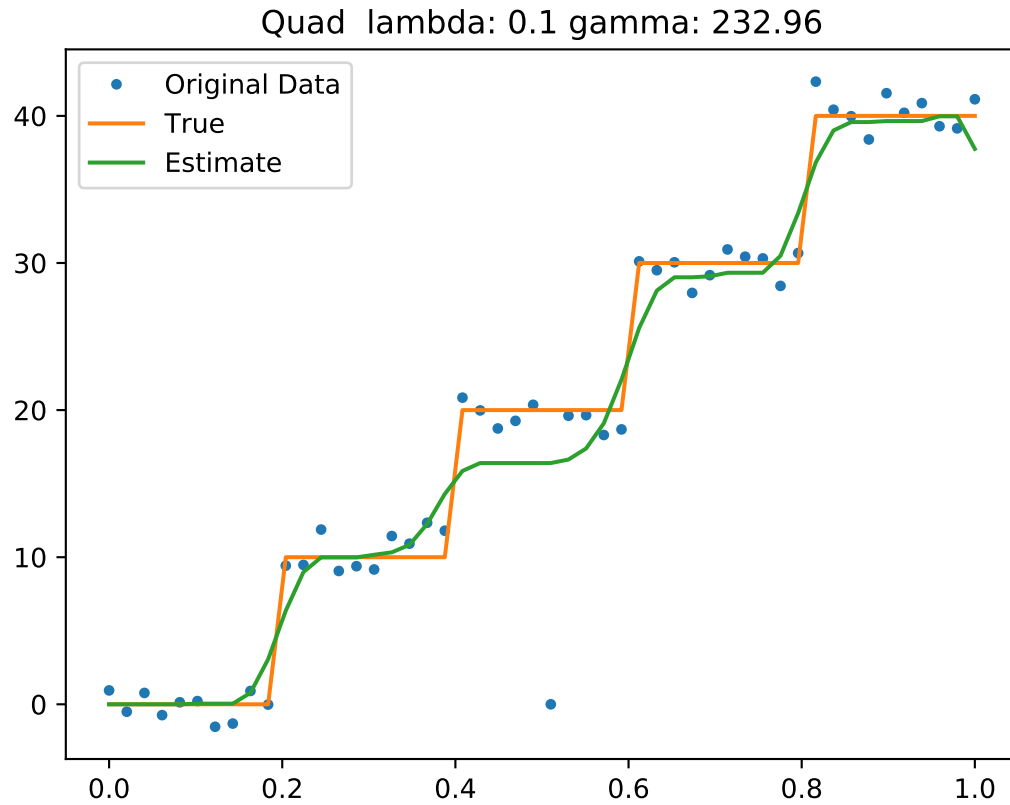
- d. We say a function g over $\{1, \dots, n\}$ is non-decreasing if $g_i - g_{i-1} \geq 0$ for all i , or $Dg \geq 0$ where the inequality applies elementwise. Perhaps due to domain knowledge, we know that the original unnoisy function we are trying to estimate is non-decreasing. Let

$$\hat{\alpha} = \arg \min_{\alpha} \|K\alpha - y\|^2 + \lambda \alpha^T K \alpha, \quad \hat{f}(x) = \sum_{i=1}^n \hat{\alpha}_i k(x_i, x)$$

subject to $DK\alpha \geq 0$

where $K_{i,j} = k(x_i, x_j)$ is a kernel evaluation and λ is a regularization constant. The above is known as a quadratic program because it can be written as $\arg \min_x \frac{1}{2} x^T Q x + p^T x + c$ subject to $Ax \leq b$. On your own (not necessary to report plots), plot \hat{f} for a variety values of γ, λ to see how they affect the solution. Use leave-one-out cross validation to find a good setting of γ, λ . Plot the original data $\{(x_i, y_i)\}_{i=1}^n$, the true $f(x)$, the $\hat{f}(x)$ found through leave-one-out CV. Note that the defined constraint only forces \hat{f} to be monotonic on the training data, not over all $x \in [0, 1]$, but it is instructive to think about how one might achieve this.

Answer:



Code:

```
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt

def generateData(n):

    i = np.array(range(1, n+1))
    x = (i-1)/(n-1)

    y = np.zeros(n)
    for i in range(1, 5):
        y += 10*(np.array(x > i/5, dtype="float"))

    f = np.copy(y)

    y += np.random.randn(n)

    y[25] = 0

    return x, y, f

def kernel(x, z, gam):

    K = np.zeros((x.size, z.size))
```

```

    for i in range(0, x.size):
        for j in range(0, z.size):
            K[i, j] = np.exp(-gam*(x[i]-z[j])**2)

    return K

def calcD(n):

    D = np.zeros((n, n-1))

    for i in range(0, n):
        for j in range(0, n-1):
            if i == j:
                D[i, j] = 1

            if i == j-1:
                D[i, j] = -1
    return D.T

def leave1(x, y):

    lamb = cp.Parameter(nonneg=True)
    # lamb2 = cp.Parameter(nonneg=True)
    lamb2Best=0
    # lambList = np.array(range(10, 20))/100
    # gamList = np.array(range(40, 100, 10))
    lambList = np.random.uniform(1, 10, 10)/100
    lamb2List = np.random.uniform(1, 10, 1)/100
    gamList = np.random.uniform(150, 250, 10)

    lambList.sort()
    gamList.sort()

    errList = np.zeros((lambList.size, gamList.size))
    err2List = np.zeros((lambList.size, lamb2List.size))

    errBest=10**100
    for gamIndex in range(gamList.size):
        print(gamIndex)
        for lambIndex in range(lambList.size):
            # for lamb2Index in range(lamb2List.size):
            errMean=0
            for i in range(0, y.size-1):
                lamb.value = lambList[lambIndex]
                # lamb2.value = lamb2List[lamb2Index]

                K = kernel(np.delete(x, i), np.delete(x, i), gamList[gamIndex])
                D = calcD(K.shape[1])

                a = cp.Variable(K.shape[1])

```

```

err = cp.sum_squares(y[i] - np.sum(K[:, i] * a))

# objective = cp.Minimize(cp.sum_squares(np.delete(y, i) - np.sum(K * a)) +
# objective = cp.Minimize(cp.sum(cp.huber(np.delete(y, i) - np.sum(K * a)))
# objective = cp.Minimize(
#     cp.sum_squares(np.delete(y, i) - np.sum(K * a)) +
#     lamb * cp.quad_form(a, K) + lamb2 * cp.norm1(D * (K * a)))
objective = cp.Minimize(cp.sum_squares(np.delete(y, i) - np.sum(K * a)) + la
constraints = [cp.matmul(cp.matmul(D, K), a) >= 0]

# prob = cp.Problem(objective)
prob = cp.Problem(objective, constraints)

prob.solve()

errMean += err.value

if errMean < errBest:
    lambBest = lambList[lambIndex]
    # lamb2Best = lambList[lamb2Index]
    gamBest = gamList[gamIndex]
    errBest = errMean

errList[lambIndex, gamIndex] = errMean
# err2List[lambIndex, lamb2Index] = errMean


plt.figure()
plt.imshow(errList)
plt.ylabel('lambda')
plt.xlabel('gamma')
plt.draw()

plt.figure()
plt.imshow(err2List)
plt.ylabel('lambda2')
plt.xlabel('lambda1')
plt.draw()

return lambBest, lamb2Best, gamBest

n = 50

x, y, f = generateData(n)

D = calcD(n)

lamb, lamb2, gam = leave1(x, y)

K = kernel(x, x, gam)

a = cp.Variable(K.shape[1])

err = cp.sum_squares(y - np.sum(K * a))

```

```

# objective = cp.Minimize(cp.sum_squares(y - np.sum(K * a)) + lamb * cp.quad_form(a, K))
# objective = cp.Minimize(cp.sum(cp.huber(y - np.sum(K * a))) + lamb * cp.quad_form(a, K))
# objective = cp.Minimize(
#     cp.sum_squares(y - np.sum(K * a)) +
#     lamb * cp.quad_form(a, K) + lamb2 * cp.norm1(D * (K * a)))
objective = cp.Minimize(cp.sum_squares(y - np.sum(K * a)) + lamb * cp.quad_form(a, K))
constraints = [cp.matmul(cp.matmul(D, K), a) >= 0]

# prob = cp.Problem(objective)
prob = cp.Problem(objective, constraints)

prob.solve()

plt.figure()
plt.plot(x, y, '. ')
plt.plot(x, f)
plt.plot(x, np.dot(K, a.value))
plt.legend(('Original_Data', 'True', 'Estimate'))
plt.title('Quad_ '+'_lambda:_' +str(round(lamb,2)) + '_gamma:_' +str(round(gam,2))) #lambda1: '+' +str(round(lamb,2))
plt.draw()
plt.savefig('2_quad.pdf')

print('Lambda_' +str(lamb))
print('Gamma_' +str(gam))
plt.show()

```

Deep learning architectures

3. *[14 points]* In this problem we will explore different deep learning architectures for a classification task. Go to http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html and complete the following tutorials

- *What is PyTorch?*
- *Autograd: automatic differentiation*
- *Neural Networks*
- *Training a classifier*

The final tutorial will leave you with a network for classifying the CIFAR-10 dataset, which is where this problem starts. Just following these tutorials could take a number of hours but they are excellent, so start early. After completing them, you should be familiar with tensors, two-dimensional convolutions (`nn.Conv2d`) and fully connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`); if there is any doubt of their inputs/outputs or whether the layers include an offset or not, consult the API <http://pytorch.org/docs/master/>.

A few preliminaries:

- Using a GPU may considerably speed up computations but it is not necessary for these small networks (one can get away with using their laptop).
- Conceptually, each network maps an image $x^{in} \in \mathbb{R}^{32 \times 32 \times 3}$ (3 channels for RGB) to an output layer $x^{out} \in \mathbb{R}^{10}$ where the image's class label is predicted as $\arg \max_{i=0,1,\dots,9} x_i^{out}$. An error occurs if the predicted label differs from its true label.

- In this problem, the network is trained via cross-entropy loss, the same loss we used for multi-class logistic regression. Specifically, for an input image and label pair (x^{input}, c) where $c \in \{0, 1, \dots, 9\}$, if the network's output layer is $x^{out} \in \mathbb{R}^{10}$, the loss is $-\log(\frac{\exp(x_c^{out})}{\sum_{c'=0}^9 \exp(x_{c'}^{out})})$.
- For computational efficiency reasons, this particular network considers *mini-batches* of images per training step meaning the network actually maps $B = 4$ images per feed-forward so that $\tilde{x}^{in} \in \mathbb{R}^{B \times 32 \times 32 \times 3}$ and $\tilde{x}^{out} \in \mathbb{R}^{B \times 10}$. This is ignored in the network descriptions below but it is something to be aware of.
- The cross-entropy loss for a neural network is, in general, non-convex. This means that the optimization method may converge to different *local minima* based on different hyperparameters of the optimization procedure (e.g., stepsize). Usually one can find a good setting for these hyperparameters by just observing the relative progress of training over the first epoch or two (how fast is it decreasing) but you are warned that early progress is not necessarily indicative of the final convergence value (you may converge quickly to a poor local minima whereas a different step size could have poor early performance but converge to a better final value).
- The training method used in this example uses a form of stochastic gradient descent (SGD) that uses a technique called *momentum* which incorporates scaled versions of previous gradients into the current descent direction². Practically speaking, momentum is another optimization hyperparameter in addition to the step size.
- We will not be using a validation set for this exercise. Hyperparameters like network architecture and step size should be chosen based on the performance on the test set. This is very bad practice for all the reasons we have discussed over the quarter, but we aim to make this exercise as simple as possible.
- You should modify the training code such that at the end of each epoch (one pass over the training data) compute and print the training and test classification accuracy (you may find the running calculation that the code initially uses useful to calculate the training accuracy).
- While one would usually train a network for hundreds of epochs for it to converge, this can be prohibitively time consuming so feel free to train your networks for just a dozen or so epochs.

You will construct a number of different network architectures and compare their performance. For all, it is highly recommended that you copy and modify the existing (working) network you are left with at the end of the tutorial *Training a classifier*. For all of the following perform a hyperparameter selection (manually by hand, random search, etc.) using the test set, report the hyperparameters you found, and plot the training and test classification accuracy as a function of iteration (one plot per network). **You will receive less credit for very sub-optimal hyperparameter choices that lead to drastically lower error rates than your peers.**

- Fully connected output, 0 hidden layers (logistic regression): we begin with the simplest network possible that has no hidden layers and simply linearly maps the input layer to the output layer. That is, conceptually it could be written as

$$x^{out} = W \text{vec}(x^{in}) + b$$

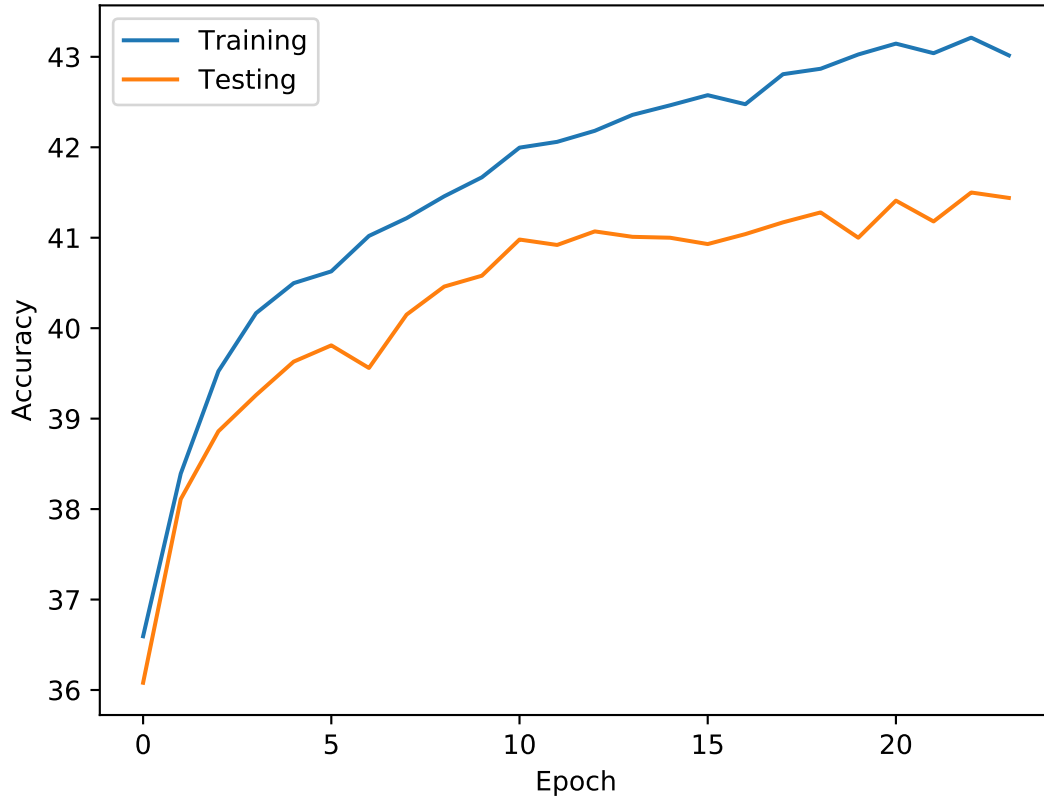
where $x^{out} \in \mathbb{R}^{10}$, $x^{in} \in \mathbb{R}^{32 \times 32 \times 3}$, $W \in \mathbb{R}^{10 \times 3072}$, $b \in \mathbb{R}^{10}$ where $3072 = 32 \cdot 32 \cdot 3$. For a tensor $x \in \mathbb{R}^{a \times b \times c}$, we let $\text{vec}(x) \in \mathbb{R}^{abc}$ be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).

Answer:

momentum: 0.05

learning rate: 0.001

²See <http://www.cs.toronto.edu/~hinton/absps/momentum.pdf> for the deep learning perspective on this method.



- b. Fully connected output, 1 fully connected hidden layer: we will have one hidden layer denoted as $x^{hidden} \in \mathbb{R}^M$ where M will be a hyperparameter you choose (M could be in the hundreds). The nonlinearity applied to the hidden layer will be the relu ($\text{relu}(x) = \max\{0, x\}$, elementwise). Conceptually, one could write this network as

$$x^{out} = W_2 \text{relu}(W_1 \text{vec}(x^{in}) + b_1) + b_2$$

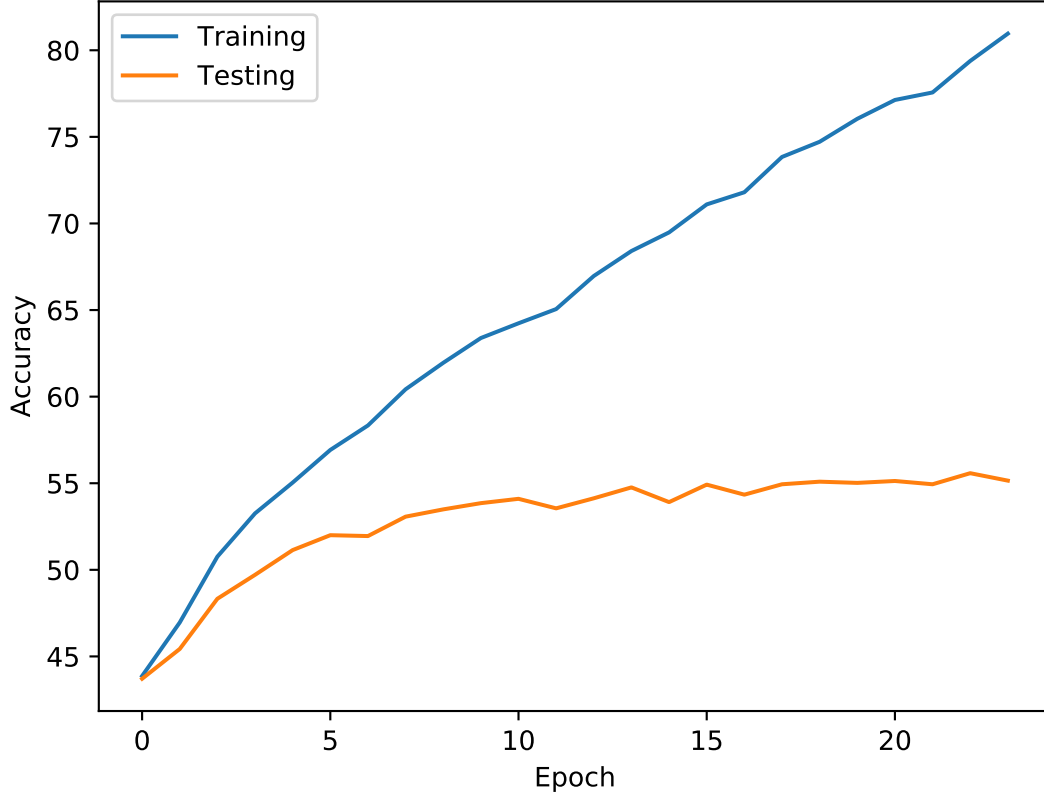
where $W_1 \in \mathbb{R}^{M \times 3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10 \times M}$, $b_2 \in \mathbb{R}^{10}$.

Answer:

momentum: 0.05

learning rate: 0.001

M=1000



- c. Fully connected output, 1 convolutional layer with max-pool: for a convolutional layer W_1 with individual filters of size $p \times p \times 3$ and output size M (reasonable choices are $M = 100$, $p = 5$) we have that $\text{Conv2d}(x^{\text{input}}, W_1) \in \mathbb{R}^{(33-p) \times (33-p) \times M}$. Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{\text{input}}, W) + b_1$ where b_1 is parameterized in \mathbb{R}^M . We will then apply a relu (relu doesn't change the tensor shape) and pool. If we use a max-pool of size N (a reasonable choice is $N = 14$ to pool to 2×2 with $p = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1)) \in \mathbb{R}^{\lfloor \frac{33-p}{N} \rfloor \times \lfloor \frac{33-p}{N} \rfloor \times M}$. We will then apply a fully connected layer to the output to get a final network given as

$$x^{\text{output}} = W_2 \text{vec}(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1))) + b_2$$

where $W_2 \in \mathbb{R}^{10 \times M(\lfloor \frac{33-p}{N} \rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$. The parameters M, p, N (in addition to the step size and momentum) are all hyperparameters.

Answer:

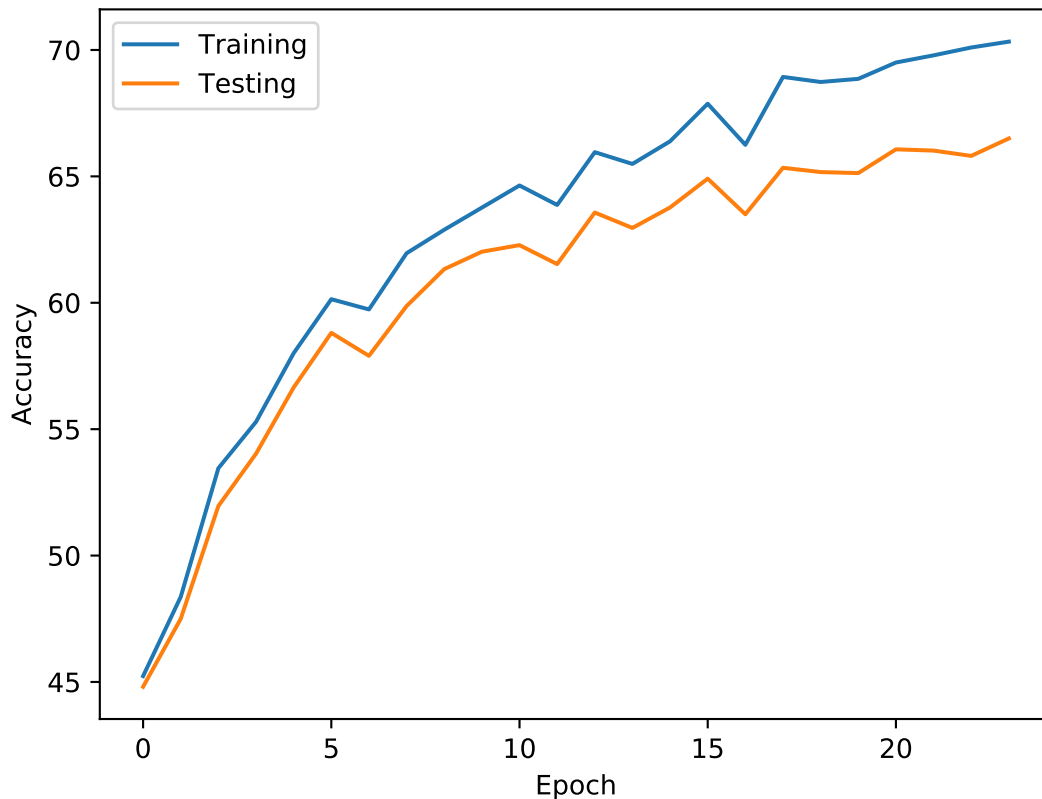
momentum: 0.05

learning rate: 0.001

$M = 1000$

$p = 5$

$N = 14$



- d. (Extra credit: *[3 points]*) Returning to the original network you were left with at the end of the tutorial *Training a classifier*, tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully connected layers, stepsize, etc.) and train for many epochs to achieve a *test accuracy* of at least 87%.

Code:

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# functions to show an image

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

class baseNet(nn.Module):
    def __init__(self):
```

```

    super(baseNet, self).__init__()
    self.conv1 = nn.Conv2d(3, 100, 5)
    self.pool = nn.MaxPool2d(2)
    self.conv2 = nn.Conv2d(100, 100, 5)
    self.fc1 = nn.Linear(100 * 5 * 5, 64)
    self.fc2 = nn.Linear(64, 32)
    self.fc3 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 100 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class logRegNet(nn.Module):
    def __init__(self):
        super(logRegNet, self).__init__()
        self.fc1 = nn.Linear(3052, 10)

    def forward(self, x):
        x = x.view(4, -1)
        x = self.fc1(x)
        return x

class singleHidNet(nn.Module):

    def __init__(self):
        super(singleHidNet, self).__init__()

        M = 1000 #100

        self.fc1 = nn.Linear(3052, M)
        self.fc2 = nn.Linear(M, 10)

    def forward(self, x):
        x = x.view(4, -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class convNet(nn.Module):

    M = 200
    p = 5
    N = 14

    def __init__(self):
        super(convNet, self).__init__()

```

```

self.conv1 = nn.Conv2d(3, self.M, self.p)
self.pool = nn.MaxPool2d(self.N, self.N)
self.fc1 = nn.Linear(int(self.M*(33-self.p)**2/self.N**2), 10)

def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = x.view(-1, int(self.M*(33-self.p)**2/self.N**2))
    x = self.fc1(x)
    return x

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                         download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                           shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                         download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                          shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

net = baseNet()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.0001, momentum=0.1)

epochLim = 25

testAcc = np.zeros(epochLim)
trainAcc = np.zeros(epochLim)

for epoch in range(epochLim): # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

```

```

    # print statistics
    running_loss += loss.item()
    if i % 2000 == 1999: # print every 2000 mini-batches
        print('[%d, %5d]_loss: %.3f' %
              (epoch + 1, i + 1, running_loss / 2000))
        running_loss = 0.0

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

print('Test accuracy: %d%%' % (
    100 * correct / total))
testAcc[epoch] = 100 * correct / total

correct = 0
total = 0
with torch.no_grad():
    for data in trainloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Train accuracy: %d%%' % (
    100 * correct / total))

trainAcc[epoch] = 100 * correct / total

print('Finished Training')

plt.figure()
plt.plot(range(epochLim), trainAcc)
plt.plot(range(epochLim), testAcc)
plt.legend(('Training', 'Testing'))
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
# plt.savefig('3_convNet.pdf')
plt.show()

```

The number of hyperparameters to tune in the last exercise combined with the slow training times hopefully gave you a taste of how difficult it is to construct good performing networks. It should be emphasized the networks we constructed are **tiny**; typical networks have dozens of layers, each with hyperparameters to tune. Additional hyperparameters you are welcome to play with if you are so interested: replacing $\text{relu max}\{0, x\}$ with a sigmoid $1/(1 + e^{-x})$, max-pool with average-pool, and experimenting with batch-normalization or dropout.