CSE 546 Homework #4

Andy Goldschmidt
12/04/2018

Expectation Maximization

1. [3 points] Pandora *is a streaming music company like* Spotify *that was known to buck the collaborative filtering trend*[1] *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 $\widehat{w} = \arg\max_w \prod_{i \in \mathcal{S}} p(y_i|x_i, w)$. The idea is that you would then construct a playlist out of the items $i \in \{1, \dots, n\}$ that maximize $x_i^T \widehat{w}$.

a. *The estimate $\widehat{w}$ has no closed-form solution. Can the optimization problem be transformed into a* convex *optimization problem? If so, suggest a method of solving for $\widehat{w}$ given $\{(x_i, y_i)\}_{i \in \mathcal{S}}$. (Hint: how do you show a function $f(x)$ is convex?).*

b. *You solve for the $\widehat{w}$ 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.*

(a.) Let $\widehat{w} := \arg\max_w \ell(w|D) = \arg\max_w \prod_{i \in \mathcal{S}} \ell_i(w|D)$ be the same maximum likelihood estimator where the dependence of the loss on w has been emphasized and the dependence on the data $D = \{(x_i, y_i)\}_{i \in \mathcal{S}}$ relegated to a parameter. The logarithm is a monotonic function and will preserve the ordering in w for the loss function. Compute:

$$\log \prod_{i \in \mathcal{S}} \ell_i(w|D) = \sum_{i \in \mathcal{S}} y_i x_i^T w - e^{w^T x_i} + \text{const.}$$

The first term $y_i x_i^T w$ is a line and so is convex. In fact, the linear case saturates the inequality in the definition of convexity such that this first term is both concave and convex. To address the second term $-e^{w^T x_i}$, compute:

$$\partial_w^2 \left(e^{w^T x_i}\right) = x_i x_i^T e^{w^T x_i} \succ 0$$

where we recall that $x_i x_i^T$ is positive semi-definite and note $e^{w^T x_i}$ is a positive scalar. Noting the minus sign, we therefore conclude the second term is concave. To undo the convavity, the problem can be transformed by including an overall minus sign on the logarithm to provide our desired concave function in $w : -\log(\ell(w))$

(b.) Let us take our k music moods to have weight vectors $\{w_k\}$. The EM Algorithm for predicting the k playlists will proceed similar to a Gaussian Mixture Model with k Gaussian distributions, except:

[1]Methods 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.

- In the "Expectation" step, the probability distribution is not a Gaussian but instead:

$$p(x_i, y_i|w_k) = \frac{\lambda_i^{y_i}}{y_i!} e^{-\lambda_i} = \frac{e^{y_i x_i^T w_k}}{y_i!} e^{-e^{w_k^T x_i}}$$

- In the "Maximization" step, the maximum likelihood estimator w_k will be computed using the convex repackaging from part a:

$$\begin{aligned} \widehat{w} &= \arg\max_w \prod_{i \in \mathcal{S}} p(x_i, y_i|w) && (1) \\ &\mapsto \arg\max_w \Big(- \log \prod_{i \in \mathcal{S}} p(x_i, y_i|w) \Big) && (2) \\ &= \arg\max_w \Big(- \sum_{i \in \mathcal{S}} \log p(x_i, y_i|w) \Big) && (3) \end{aligned}$$

The algorithm follows:

EM Algorithm for k playlists

I Randomly assign w_k.

II Compute $p(w_k|x_i, y_i)$ $\forall k, i$ by using the Bayesian prior distribution we defined in part a. Assume $p(w_k)$ are the same $\forall k$–this could be changed if certain moods are more prevalent.

III Update

$$w_k \leftarrow \arg\max_w \left(\frac{- \sum_{i \in \mathcal{S}} \log(p(x_i, y_i|w)) p(w_k|x_i, y_i)}{\sum_{i \in \mathcal{S}} p(w_k|x_i, y_i)} \right)$$

This is the weighted average of the loss according to the probability of w_k given data i (computed in the previous step).

IV Repeat the previous two steps until convergence.

The result of the algorithm are weight vectors w_k that are the moods of the person. These w_k assign a probability over the songs and listening frequencies in $\{(x_i, y_i)\}$. We can build a playlist for mood k at random by adding songs according to some score defined by $w_k^T x$. For example, we might take the expected number of listens to define a probability distribution on all songs as $\frac{\exp\{w_k^T x\}}{\sum_x \exp\{w_k^T x\}}$ and use this to generate the playlist.

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 $f(x) = 10 \sum_{k=1}^{4} \mathbb{1}\{x \geq \frac{k}{5}\}$, noting that $f(x)$ is non-decreasing in x (i.e., $f(x) \geq f(z)$ whenever $x \geq z$). For $i = 1, \ldots, n$ let each $x_i = \frac{i-1}{n-1}$ and $y_i = \mathbb{1}\{i \neq 25\}(f(x_i) + \epsilon_i)$ where $\epsilon_i \sim \mathcal{N}(0, 1)$. 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:*

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

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*

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

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* $\widehat{f}(x)$ *found through leave-one-out CV. (Hint: start with the problem on the homepage of* ***http://www.cvxpy.org/*** *and modify it as needed.)*

b. *Now let*

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

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* $\widehat{f}(x)$ *found through leave-one-out CV. (Hint:* ***huber*** *is a function in* ***cvxpy****.)*

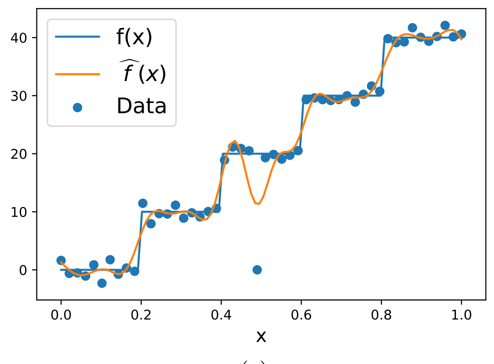
c. *The total variation (TV) of a real-valued function* g *over* $\{1,\dots,n\}$ *is defined as* $\int_{i=1}^{n-1} |g_i - g_{i-1}| = ||Dg||_1$ *and is a common regularizer for de-noising a function (its two-dimensional counterpart is very popular for image de-noising or filling-in missing/damaged parts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

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

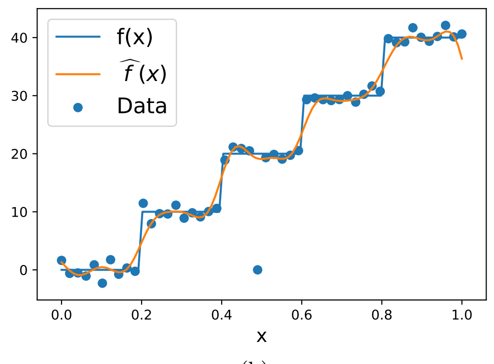
where $K_{i,j} = k(x_i,x_j)$ *is a kernel evaluation and* λ_1, λ_2 *are regularization constants. For intuition, the penalizer* $||DK\alpha||_1$ *prefers functions* $\widehat{f}$ *with sparse jumps in function value while* $\alpha^T K\alpha$ *perfers functions that are smoothly varying. On your own (not necessary to report plots), plot* $\widehat{f}$ *for a variety 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* $\widehat{f}(x)$ *found through leave-one-out CV.*

d. *We say a function* g *over* $\{1,\dots,n\}$ *is non-decreasing if* $g_i - g_{i-1} \ge 0$ *for all* i, *or* $Dg \ge 0$ *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*

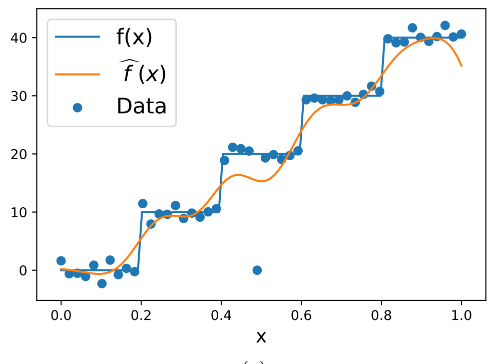
$$\widehat{\alpha} = \arg\min_\alpha ||K\alpha - y||^2 + \lambda\alpha^T K\alpha \ , \qquad \widehat{f}(x) = \sum_{i=1}^n \widehat{\alpha}_i k(x_i,x)$$
$$\text{subject to} \quad DK\alpha \ge 0$$



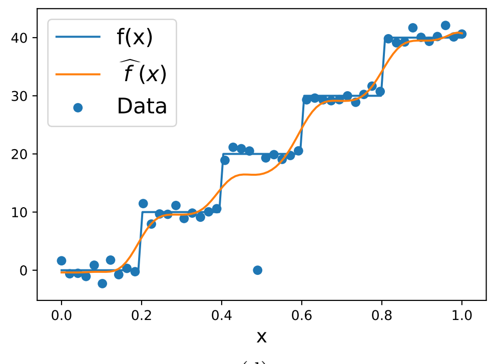
(a)



(b)



(c)



(d)

Figure 1: Sub-figures 1(a)-1(d) correspond to their respective parts of Problem 2. Each $\widehat{f}(x)$ is a solution to an optimization problem solved for the same data. The data are generated from an exact solution that has been plotted as $f(x)$. Notice the outlier at $x \sim 0.5$: the optimization problems show the resiliance of different methods to an outlier.

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 $\arg\min_x \frac{1}{2}x^T Qx + p^T x + c$ subject to $Ax \leq b$. On your own (not necessary to report plots), plot $\widehat{f}$ 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 $\widehat{f}(x)$ found through leave-one-out CV. Note that the defined constraint only forces $\widehat{f}$ 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.

The code for Problem 2 is in Listing 1. The requested plots for parts (a)-(d) are in Figure 1. Hyper-parameters were found using leave-one-out cross-validation over ∼100 iterations of a random search. The results are reported in Table 1.

Table 1: $\widehat{f}$ hyper-parameters

	λ (quadratic form K)	γ (rbf kernel)	M (for ℓ_{Huber})	λ_{TV} (total variation)
(a)	0.0013	179	-	-
(b)	0.0021	211	0.54	-
(c)	0.32	86.5	-	1.13
(d)	0.090	128	-	-

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 $-\log(\frac{\exp(x_c^{out})}{\sum_{c'=0}^{9} x_{c'}^{out}})$.*
- *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 $\widetilde{x}^{in} \in \mathbb{R}^{B\times 32\times 32\times 3}$ and $\widetilde{x}^{out} \in \mathbb{R}^{B\times 10}$. 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 direction*[2]*. 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.*

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

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.***

a. *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*

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

where $x^{out} \in \mathbb{R}^{10}$, $x^{in} \in \mathbb{R}^{32\times32\times3}$, $W \in \mathbb{R}^{10\times3072}$, $b \in \mathbb{R}^{10}$ *where* $3072 = 32 \cdot 32 \cdot 3$. *For a tensor* $x \in \mathbb{R}^{a\times b\times c}$, *we let* $\vec{(}x) \in \mathbb{R}^{abc}$ *be the reshaped form of the tensor into a vector (in an arbitrary but consistent pattern).*

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*

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

where $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$.

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* $\text{Conv2d}(x^{input}, W_1) \in \mathbb{R}^{(33-p)\times(33-p)\times M}$. *Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as* $\text{Conv2d}(x^{input}, W) + b_1$ *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* $MaxPool(\text{relu}(\text{Conv2d}(x^{input}, W_1) + b_1)) \in \mathbb{R}^{\lfloor\frac{33-p}{N}\rfloor\times\lfloor\frac{33-p}{N}\rfloor\times M}$. *We will then apply a fully connected layer to the output to get a final network given as*

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

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.*

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%.*

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 relu* $\max\{0, x\}$ *with a sigmoid* $1/(1 + e^{-x})$, *max-pool with average-pool, and experimenting with batch-normalization or dropout.*

The code for Problem 3 is in Listing 2. The requested plots for parts (a)-(c) are in Figure 2, where the training and test accuracy are plotted for two sets of hyper-parameters (labelled "Bad" and "Good" according to their relative convergence rates) for each network. Note that the difference between bad and good is mostly an order-of-magnitude increase in the gradient descent momentum hyper-parameter. Sometimes the resulting difference in convergence isn't so dramatic, but we'll stick with good/bad labels for convenience. Hyper-parameters were found by graduate student descent (using the hints provided in the problem statement). The hyper-parameter results are reported in Table 2.

The peak accuracies are:

a. Problem 3a's network: 41.22%

b. Problem 3b's network: 52.52%

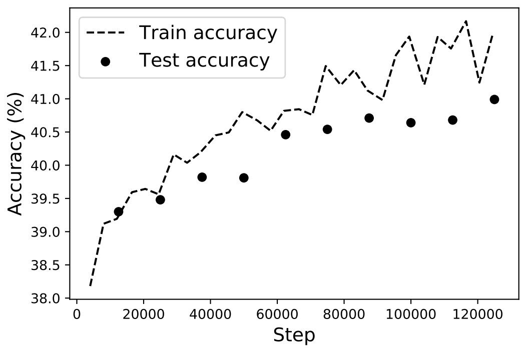
c. Problem 3c's network: 68.09%

Table 2: Network hyper-parameters

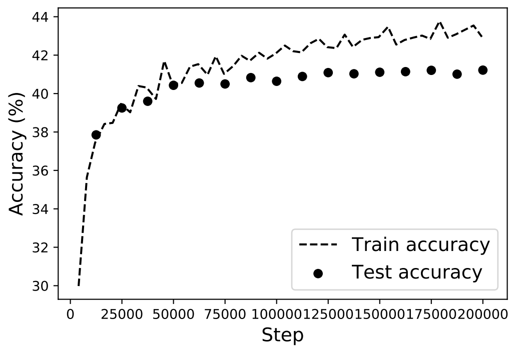
	epochs	learning rate	momentum	M	p	N
Problem 3a "Bad"	10	0.0001	0.01	-	-	-
Problem 3a "Good"	16	0.0001	0.5	-	-	-
Problem 3b "Bad"	16	0.0001	0.01	150	-	-
Problem 3b "Good"	16	0.001	0.7	150	-	-
Problem 3c "Bad"	16	0.0001	0.01	150	6	4
Problem 3c "Good"	16	0.001	0.7	150	6	4

Table 3: The variable names are chosen to match the language of the problem statement. For specific information about each parameter, consult the appropriate part of Problem 3.

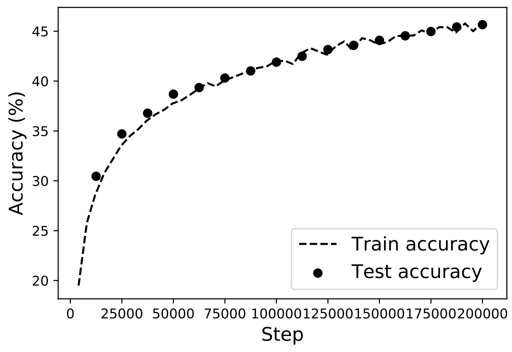
Note that for effiency, the training accuracy in Figure 1 is found by a "running" calculation of the accuracy within the training loop. This results in a sort of "average" accuracy being reported, but the appropriate trends are captured. The test accuracy is computed in the expected way. (See Listing 2 for more).



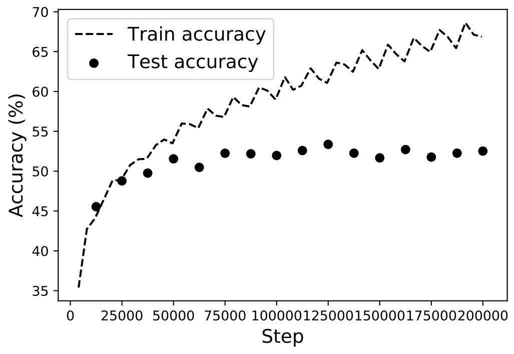
(a) Problem 3a: Bad convergence



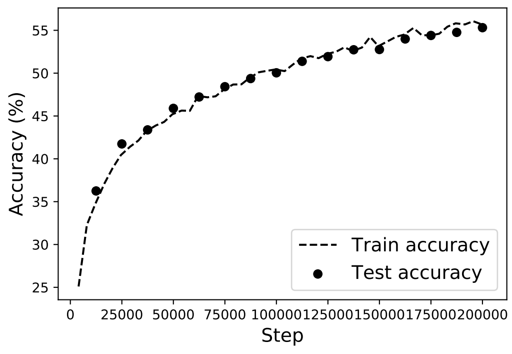
(b) Problem 3a: Good convergence



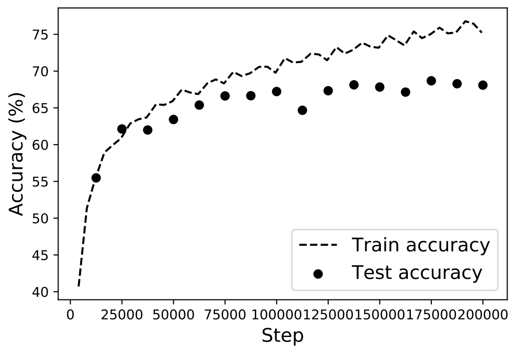
(c) Problem 3b: Bad convergence



(d) Problem 3b: Good convergence



(e) Problem 3c: Bad convergence



(f) Problem 3c: Good convergence

Figure 2: Sub-figures 2(a)-2(b) correspond to the network in Problem 3a, 2(c)-2(d) correspond to the network in Problem 3b, and 2(d)-2(e) correspond to the network in Problem 3c. Each shows the training and test error for the network as a function of training iteration. Plots on the left show "bad" hyper-parameter choices, while plots on the right show "good" hyper-parameter choices. Exact parameters can be found in Table 2. A single training epoch is 12500 iterations.

Listing 1:

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
from itertools import compress, combinations

# ------------------------------------------------------------------------

def function(x):
    '''
    Array-friendly 4-step function on [0,1.]
    '''
    return 10*np.sum([x >= k/5 for k in [1,2,3,4]], axis=0)

def k_rbf(x,z,gamma):
    '''
    rbf (gaussian kernel)
    '''
    return np.exp(-gamma*np.square(x-z))

def ls_loss_fn(residuals, params=None):
    return cp.pnorm(residuals, p=2)**2

def huber_loss_fn(residuals, params):
    M = params
    return np.sum([i for i in cp.huber(residuals, M)])

def regularizer(alpha, K):
    return cp.quad_form(alpha, K)


# ------------------------------------------------------------------------
# Lazy 3 functions that do basically the same thing:
# Solve the optimization problem
# ------------------------------------------------------------------------

def cvx_kernel_1d(x, y, loss_fn, loss_params, kernel, k_params, lambd):
    '''
    Return function that computes sum over all alpha-weighted kernel data
        vectors
    for each new x (each x_p in possible many x_predict).

    Notes on cvxpy:
    (1) The optimal objective value is returned by `cp.Problem.solve()`.
    (2) After solve, optimal value for alpha is stored in `alpha.value`.

    @requires loss_fn take args ([K,alpha,y],*loss_params)
    Expect that x_predict is an iterable of x_p data
    '''
    xi, xj = np.meshgrid(x, x)
    K = kernel(xi, xj, k_params)

    alpha = cp.Variable(len(x))
    _lambd = cp.Parameter(nonneg=True)
    _lambd.value = lambd
    residuals = cp.matmul(K, alpha) - y
    objective = cp.Minimize(loss_fn(residuals, loss_params) + _lambd*regularizer
        (alpha, K))
    prob = cp.Problem(objective)
    prob.solve()

    return lambda x_predict: np.array([np.sum(alpha.value*kernel(x, x_p,
        k_params))
                                      for x_p in x_predict])


def cvx_kernel_1d_type2(x, y, loss_fn, loss_params, kernel, k_params, lambd1,
    lambd2):
    xi, xj = np.meshgrid(x, x)
    K = kernel(xi, xj, k_params)

    n = len(x)
    alpha = cp.Variable(n)
    D = -1*np.identity(n)[:-1] + np.identity(n)[1:]
    _lambd1 = cp.Parameter(nonneg=True)
    _lambd2 = cp.Parameter(nonneg=True)
    _lambd1.value = lambd1
    _lambd2.value = lambd2

    residuals = cp.matmul(K, alpha) - y
    objective = cp.Minimize(loss_fn(residuals, loss_params)
                            + _lambd1 * cp.norm1(cp.matmul(D, cp.matmul(K, alpha
                                )))
                            + _lambd2 * regularizer(alpha, K))
    prob = cp.Problem(objective)
    prob.solve()

    return lambda x_predict: np.array([np.sum(alpha.value*kernel(x, x_p,
        k_params))
                                      for x_p in x_predict])


def cvx_kernel_1d_type3(x, y, loss_fn, loss_params, kernel, k_params, lambd):
    xi, xj = np.meshgrid(x, x)
    K = kernel(xi, xj, k_params)

    n = len(x)
    D = -1*np.identity(n)[:-1] + np.identity(n)[1:]
    alpha = cp.Variable(n)
    _lambd = cp.Parameter(nonneg=True)
    _lambd.value = lambd

    residuals = cp.matmul(K, alpha) - y
    objective = cp.Minimize(loss_fn(residuals, loss_params) + _lambd*regularizer
        (alpha, K))
    constraints = [cp.matmul(D, cp.matmul(K, alpha)) >= 0]
    prob = cp.Problem(objective, constraints)
    prob.solve()
    return lambda x_predict: np.array([np.sum(alpha.value*kernel(x, x_p,
        k_params))
                                        for x_p in x_predict])

# ------------------------------------------------------------------------
# k_fold for each function
# ------------------------------------------------------------------------
def k_fold_cv(k, x, y, loss_fn, loss_params, kernel, k_params, lambd):
    '''
    Returns the average loss of the k-fold cross validation.
    @requires len(x)/k in integers (creates len(x)/k folds)
    '''
    n = len(x)
    indices = np.arange(n).astype(int)
    k_folds = np.random.permutation(indices).reshape(int(n/k), k) # Each row is
        a k-fold.
    k_loss = np.zeros(int(n/k))
    for i, k_validation in enumerate(k_folds):
        select_train = np.ones(n).astype(int)
        select_train[k_validation] = 0
        x_train = np.array([xi for xi in compress(x, select_train)])
        y_train = np.array([yi for yi in compress(y, select_train)])

        f_hat_k = cvx_kernel_1d(x_train, y_train, loss_fn, loss_params, kernel,
            k_params, lambd)

        residuals = y[k_validation]-f_hat_k(x[k_validation])
        k_loss[i] = (loss_fn(residuals, loss_params)/len(k_validation)).value
    return np.mean(k_loss)

def k_fold_cv_type2(k, x, y, loss_fn, loss_params, kernel, k_params, lambd1,
   lambd2):
    '''
    Returns the average loss of the k-fold cross validation.
    @requires len(x)/k in integers (creates len(x)/k folds)
    '''
    n = len(x)
    indices = np.arange(n).astype(int)
    k_folds = np.random.permutation(indices).reshape(int(n/k), k) # Each row is
        a k-fold.
    k_loss = np.zeros(int(n/k))
    for i, k_validation in enumerate(k_folds):
        select_train = np.ones(n).astype(int)
        select_train[k_validation] = 0
        x_train = np.array([xi for xi in compress(x, select_train)])
        y_train = np.array([yi for yi in compress(y, select_train)])

        f_hat_k = cvx_kernel_1d_type2(x_train, y_train, loss_fn, loss_params,
            kernel, k_params, lambd1, lambd2)

        residuals = y[k_validation]-f_hat_k(x[k_validation])
        k_loss[i] = (loss_fn(residuals, loss_params)/len(k_validation)).value
    return np.mean(k_loss)

def k_fold_cv_type3(k, x, y, loss_fn, loss_params, kernel, k_params, lambd):
    n = len(x)
    indices = np.arange(n).astype(int)
    k_folds = np.random.permutation(indices).reshape(int(n/k), k) # Each row is
       a k-fold.
    k_loss = np.zeros(int(n/k))
    for i, k_validation in enumerate(k_folds):
        select_train = np.ones(n).astype(int)
        select_train[k_validation] = 0
        x_train = np.array([xi for xi in compress(x, select_train)])
        y_train = np.array([yi for yi in compress(y, select_train)])

        f_hat_k = cvx_kernel_1d_type3(x_train, y_train, loss_fn, loss_params,
           kernel, k_params, lambd)

        residuals = y[k_validation]-f_hat_k(x[k_validation])
        k_loss[i] = (loss_fn(residuals, loss_params)/len(k_validation)).value
    return np.mean(k_loss)


# ==========================================================================
# Main functions: need one plot for each (a) - (d)
# We'll hardcode 100 iterations for each hyper-parameter search
# ==========================================================================
if __name__ == '__main__':
    # prelims
    n=50
    x_data = np.array([(i-1)/(n-1) for i in range(1,n+1)])
    np.random.seed(1)
    y_data = np.array([0 if i==25 else 1 for i in range(1,n+1)])*(function(
       x_data) + np.random.randn(n))


    # --- part a ---------------------------------------------------------------
    g_max = 1000
    g_min = 1

    lambd_max = 5
    lambd_min = .005

    best_error_ls = np.inf
    best_ls = []

    for i in range(100):
        gamma0 = np.random.rand()*(g_max-g_min) + g_min
        lambd0 = np.random.rand()*(lambd_max-lambd_min) + lambd_min
        error = k_fold_cv(1, x_data, y_data, ls_loss_fn, None, k_rbf, gamma0,
           lambd0)
        if error < best_error_ls:
            best_error_ls = error
            best_ls = [gamma0, lambd0]

    f_hat = cvx_kernel_1d(x_data, y_data,
                          loss_fn=ls_loss_fn, loss_params=None,
                          kernel=k_rbf, k_params=best_ls[0], lambd=best_ls[1])
```

```
fig, ax = plt.subplots(1)
ax.plot(x, function(x))
ax.scatter(x_data, y_data)
ax.plot(x, f_hat(x))
ax.legend(['f(x)', '$\widehat{\ f\ }(x)}$', 'Data'], fontsize=16)
ax.set_xlabel('x', fontsize=14)
fig.savefig('figs/hw4_2a_ls_plt.pdf')
np.savetxt(X=best_ls,fname='figs/hw4_2a_ls.txt')

# --- part b -----------------------------------------------------------------
g_max = 1000
g_min = 1

lambd_max = 5
lambd_min = .005

M_max = 10
M_min = .1

best_huber = []
best_error_huber = np.inf

for i in range(100):
    gamma0 = np.random.rand()*(g_max-g_min) + g_min
    lambd0 = np.random.rand()*(lambd_max-lambd_min) + lambd_min
    M0 = np.random.rand()*(M_max-M_min) + M_min
    error = k_fold_cv(5, x_data, y_data, huber_loss_fn, M0, k_rbf, gamma0,
        lambd0)
    if error < best_error_huber:
        best_error_huber = error
        best_huber = [gamma0, lambd0, M0]

f_hat = cvx_kernel_1d(x_data, y_data,
                      loss_fn=huber_loss_fn, loss_params=best_huber[2],
                      kernel=k_rbf, k_params=best_huber[0], lambd=best_huber
                          [1])

fig, ax = plt.subplots(1)
ax.plot(x, function(x))
ax.scatter(x_data, y_data)
ax.plot(x, f_hat(x))
ax.legend(['f(x)', '$\widehat{\ f\ }(x)}$', 'Data'], fontsize=16)
ax.set_xlabel('x', fontsize=14)
fig.savefig('figs/hw4_2b_huber_plt.pdf')
np.savetxt(X=best_huber,fname='figs/hw4_2b_huber.txt')

# --- part c -----------------------------------------------------------------
g_max = 1000
g_min = 1

lambd1_max = 5
lambd1_min = .005

lambd2_max = 5
lambd2_min = .005
```

```
best_tv = []
best_error_tv = np.inf

for i in range(100):
    gamma0 = np.random.rand()*(g_max-g_min) + g_min
    lambd1 = np.random.rand()*(lambd1_max-lambd1_min) + lambd1_min
    lambd2 = np.random.rand()*(lambd2_max-lambd2_min) + lambd2_min
    error = k_fold_cv_type2(5, x_data, y_data, ls_loss_fn, None, k_rbf,
        gamma0, lambd1, lambd2)
    if error < best_error_tv:
        best_error_tv = error
        best_tv = [gamma0, lambd1, lambd2]

f_hat = cvx_kernel_1d_type2(x_data, y_data,
                        loss_fn=ls_loss_fn, loss_params=None,
                        kernel=k_rbf, k_params=best_tv[0],
                        lambd1=best_tv[1], lambd2=best_tv[2])

fig, ax = plt.subplots(1)
ax.plot(x, function(x))
ax.scatter(x_data, y_data)
ax.plot(x, f_hat(x))
ax.legend(['f(x)', '$\widehat{\_f\_}(x)}$', 'Data'], fontsize=16)
ax.set_xlabel('x', fontsize=14)
fig.savefig('figs/hw4_2c_tv_plt.pdf')
np.savetxt(X=best_tv,fname='figs/hw4_2c_tv.txt')

# --- part d ----------------------------------------------------------------
g_max = 1000
g_min = 1

lambd_max = 5
lambd_min = .005

best_error_qp = np.inf
best_qp = []

for i in range(100):
    gamma0 = np.random.rand()*(g_max-g_min) + g_min
    lambd0 = np.random.rand()*(lambd_max-lambd_min) + lambd_min
    error = k_fold_cv_type3(1, x_data, y_data, ls_loss_fn, None, k_rbf,
        gamma0, lambd0)
    if error < best_error_qp:
        best_error_qp = error
        best_qp = [gamma0, lambd0]

f_hat = cvx_kernel_1d_type3(x_data, y_data, loss_fn=ls_loss_fn, loss_params=
    None,
                            kernel=k_rbf, k_params=best_qp[0], lambd=best_qp
                                [1])

fig, ax = plt.subplots(1)
ax.plot(x, function(x))
ax.scatter(x_data, y_data)
ax.plot(x, f_hat(x))
ax.legend(['f(x)', '$\widehat{\ f\ }(x)}$', 'Data'], fontsize=16)
ax.set_xlabel('x', fontsize=14)
fig.savefig('figs/hw4_2d_qp_plt.pdf')
np.savetxt(X=best_tv,fname='figs/hw4_2d_qp.txt')
```

Listing 2:

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

import datetime

import pickle
from os.path import join

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms as transforms
import torch.optim as optim

# Assumes there is a .obj directory in the current folder (for backup)
# https://stackoverflow.com/questions/19201290
def save_obj(obj, name, root=''):
    with open(join(root, '.obj', name + '.pkl'), 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)

def load_obj(name, root=''):
    with open(join(root, '.obj/', name + '.pkl'), 'rb') as f:
        return pickle.load(f)
# ------------------------------------------------------

def _num_flat_features(x):
    size = x.size()[1:]  # all dimensions except the batch dimension
    num_features = 1
    for s in size:
        num_features *= s
    return num_features

class Net_C(nn.Module):
    def __init__(self, M, p, N):
        '''
        M is the number of output channels, p is the convolution kernel size,
        N is the max pooling kernel (ideally, it is a divisor of 33-p)
        '''
        super(Net_C, self).__init__()
        # 3 input image channel, M output channels, pxpx3 square convolution
        # bias=True is default
        self.conv1 = nn.Conv2d(3, M, p)
        self.pool1 = nn.MaxPool2d(N)
        self.fc1 = nn.Linear(M*((33-p)//N)**2, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = x.view(-1, self.num_flat_features(x)) # 32 * 32 * 3
        x = self.fc1(x)
        return x

    def num_flat_features(self, x):
        return _num_flat_features(x)
```

```
class Net_B(nn.Module):
    def __init__(self, M):
        super(Net_B, self).__init__()
        self.fc1 = nn.Linear(32 * 32 * 3, M)
        self.fc2 = nn.Linear(M, 10)

    def forward(self, x):
        x = x.view(-1, self.num_flat_features(x)) # 32 * 32 * 3
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    def num_flat_features(self, x):
        return _num_flat_features(x)


class Net_A(nn.Module):
    def __init__(self):
        super(Net_A, self).__init__()
        self.fc1 = nn.Linear(32 * 32 * 3, 10)

    def forward(self, x):
        x = x.view(-1, self.num_flat_features(x)) # 32 * 32 * 3
        x = self.fc1(x)
        return x

    def num_flat_features(self, x):
        return _num_flat_features(x)
# ------------------------------------------------------------------

# Training an image classifier

# 1 Load and normalizing the CIFAR10 training and test datasets using
    torchvision
# 2 Define a Convolution Neural Network
# 3 Define a loss function
# 4 Train the network on the training data
# 5 Test the network on the test data

def train_net(net, iter_trainloader, iter_testloader, criterion, optimizer,
    epochs):
    '''
    Trains net, updates training error with iterations, and returns epoch-wise
        test loss
    '''
    epochs_train_error = []
    epochs_test_error = []
    for epoch in range(epochs):  # loop over the dataset multiple times
        # keep track of percentage correct
        total = 0
        correct = 0
        for i, data in enumerate(iter_trainloader):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            # (also do some percent error tallies... before optimization? sure.)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # running calc of error occurs every 4000 mini-batches
            if i % 4000 == 3999:
                epochs_train_error.append([epoch, i, 100 * correct / total])
                total = 0
                correct = 0

        # Append the test errors for this epoch
        epochs_test_error.append([epoch, 12500, test_net(net, iter_testloader,
            criterion)])
    return epochs_train_error, epochs_test_error


def test_net(net, iter_testloader, criterion):
    with torch.no_grad():
        correct = 0
        total = 0
        for data in iter_testloader:
            images, labels = data
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return 100 * correct / total
# ---------------------------------------------------------------------------

def main():
    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')

# Shared criterion for loss
criterion_all = nn.CrossEntropyLoss()

# ======== Network A
a_params = {
    'lr': 0.0001,
    'momentum': 0.01,
    'epochs': 10
}

netA = Net_A()

# Declare some optimizer for Net A
optimizerA = optim.SGD(netA.parameters(), lr=a_params['lr'], momentum=
    a_params['momentum'])

resA_train, resA_test = train_net(netA, trainloader, testloader,
    criterion_all, optimizerA, a_params['epochs'])

# Save network and parameters (name with time-stamp and percent error)
resA_train = np.array(resA_train)
resA_test = np.array(resA_test)

timestrA = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
save_obj(name='netA_{}_{}'.format(int(resA_test[-1,2]), timestrA), obj=[netA
    , resA_train, resA_test, a_params])

fig, ax = plt.subplots(1)
ax.plot(resA_train[:,0]*12500 + resA_train[:,1], resA_train[:,2], '--', c='k
    ')
ax.scatter(resA_test[:,0]*12500 + resA_test[:,1], resA_test[:,2], c='k')
ax.set_xlabel('Step', fontsize=14)
ax.set_ylabel('Accuracy␣(%)', fontsize=14)
ax.legend(['Train␣accuracy', 'Test␣accuracy'], fontsize=14)
fig.savefig('figs/hw5_p3a_{}.pdf'.format(timestrA))

print("A:␣", resA_test[-1,2])

# ======== Network B
b_params = {
    'lr': 0.001,
    'momentum': 0.7,
    'M': 150,
    'epochs': 16
}

netB = Net_B(b_params['M'])

optimizerB = optim.SGD(netB.parameters(), lr=b_params['lr'], momentum=
    b_params['momentum'])

resB_train, resB_test = train_net(netB, trainloader, testloader,
    criterion_all, optimizerB, b_params['epochs'])

# Save network and parameters (name with time-stamp and percent error)
resB_train = np.array(resB_train)
resB_test = np.array(resB_test)

timestrB = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
save_obj(name='netB_{}_{}'.format(int(resB_test[-1,2]), timestrB), obj=[netB
    , resB_train, resB_test, b_params])

fig, ax = plt.subplots(1)
ax.plot(resB_train[:,0]*12500 + resB_train[:,1], resB_train[:,2], '--', c='k
    ')
ax.scatter(resB_test[:,0]*12500 + resB_test[:,1], resB_test[:,2], c='k')
ax.set_xlabel('Step', fontsize=14)
ax.set_ylabel('Accuracy (%)', fontsize=14)
ax.legend(['Train accuracy', 'Test accuracy'], fontsize=14)
fig.savefig('figs/hw5_p3b_{}.pdf'.format(timestrB))

print("B: ", resA_test[-1,2])

# ========== Network C
c_params = {
    'lr': 0.001,
    'momentum': 0.7,
    'M': 150,
    'p': 6,
    'N': 4,
    'epochs': 16
}

netC = Net_C(c_params['M'], c_params['p'], c_params['N'])

optimizerC = optim.SGD(netC.parameters(), lr=c_params['lr'], momentum=
    c_params['momentum'])

resC_train, resC_test = train_net(netC, trainloader, testloader,
    criterion_all, optimizerC, c_params['epochs'])

# Save network and parameters (name with time-stamp and percent error)
resC_train = np.array(resC_train)
resC_test = np.array(resC_test)

timestrC = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
save_obj(name='netC_{}_{}'.format(int(resC_test[-1,2]), timestrC), obj=[netC
    , resC_train, resC_test, c_params])

fig, ax = plt.subplots(1)
ax.plot(resC_train[:,0]*12500 + resC_train[:,1], resC_train[:,2], '--', c='k
    ')
ax.scatter(resC_test[:,0]*12500 + resC_test[:,1], resC_test[:,2], c='k')
ax.set_xlabel('Step', fontsize=14)
ax.set_ylabel('Accuracy (%)', fontsize=14)
ax.legend(['Train accuracy', 'Test accuracy'], fontsize=14)
fig.savefig('figs/hw5_p3c_{}.pdf'.format(timestrC))
```

```
    print("C: ", resC_test[-1,2])


if __name__ == '__main__':
    main()
```