CS 6923 Machine Learning Final Project Report

Group members:
Ruinan Zhang *rz1109@nyu.edu*
Xianbo Gao *xg656@nyu.edu*

Spring 2020
Prof Linda Sellie

1 Introduction and Report Structure:

1.1 Introduction

In this project, we explored three ML algorithms we learned during the pass of this course, for each basic algorithm, we came up with one extension that can improve the overall performance of that algorithm. And then, we first implemented the basic and extended algorithms using existing libraries (Scikit-learn, Tensorflow,Pytorch...). Second, we built our own extension only Numpy from scratch.
The three algorithms we explored and corresponding extensions are:

1. **Support vector machine (SVM) with Soft Margin**
2. **Linear Regression vs. Lasso Regression with Coordinate descent**
3. **Neural Network with Convolutional layers**

1.2 Report Structure

This report will follow the structure listed below for each of the three extensions:

1. Explanation about the extension's algorithm
2. Summary of the extensions' implementation on Sklearn and Numpy Algo
3. Accuracies comparison
4. Explanations on specific datasets
5. NumPy code for extensions
6. Further Explanations

2 Support vector machine (SVM) with Soft Margin

2.1 Explanation about SVM with soft margin algorithm

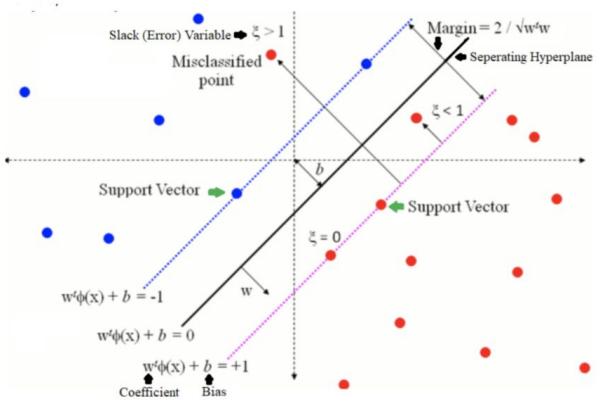
In homework9, we implemented hard margin SVM and solved both primal and dual problems. In this project, we will focus on solving soft margin dual problems as most datasets in real life are not linearly separable.
Remember in **hard margin SVM case**, constrained optimization problem **(primal)** is:

$$\begin{aligned} & min \frac{1}{2}||w||_2^2 \\ \text{s.t.} \quad & y^{(i)}(w_0 + w^T x^{(i)}) \geq 1 \\ & \text{for all i} = 1, 2....,N \end{aligned}$$

And after we transform our original problem into its **duel** formalization, adding Lagrangian multiplier, we have a dual optimization problem that is:

$$\begin{aligned} & min_\alpha \frac{1}{2}\alpha^T Q\alpha - 1^T \alpha \\ \text{s.t.} \quad & \alpha_i \geq 0 \quad \forall i \\ \text{and} \quad & y^T \alpha = 0 \end{aligned}$$

The hard margin SVM has its limit as it assumes that our data is linearly separable. But what if we have a few misclassified data or data points that within the margin?
Then we extend to the **soft margin SVM case** -- which introduces a new tolerance variable **(or slack variable) $\zeta(i)$ for each training example $x(i)$.**



[1] Pic Sourc

Points that lie on the margin will have $\zeta(i)=0$. On the other hand, the farther we are from the correct boundary, the larger the value of $\zeta(i)$ will be larger.
Thus, our optimization function would be:

$$min\frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\zeta^{(i)}$$
$$\text{s.t.} \quad y^{(i)}(w_0 + w^T x^{(i)}) \geq 1 - \zeta^{(i)}$$
$$\text{for all i} = 1, 2....,N$$
$$\zeta^{(i)} \geq 0$$

Notice here C is a tunable parameter, gives relative importance of the error term. It controls how much you want to punish your model for each misclassified point for a given curve. The lower the C parameters, the softer the margin.
After adding the Lagrangian multiplier, to do the kernel trick, then we have the SVM soft margin dual quadratic programming problem:

$$min_\alpha \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j K(x_i, x_j) - \sum_{i=1}^{m}\alpha_i$$
$$\text{s.t.} \quad C \geq \alpha_i \geq 0$$
$$\sum_{i=1}^{m}\alpha_i y_i = 0$$

Then, we use CVXOPT, the quadratic programming problem solver, cvxopt.solvers.qp to solve this problem. We can see this is very similar to the homework's problem of dual for hard margin besides we have an additional constraint on α.
When using CVXOPT's quadratic solver, we implement this constraint by concatenating below matrix G a diagonal matrix of 1s of size m×m and adding m times value to C in vector h.

2.2 Summary SVM's soft margin's implementation using Sklearn and Numpy

- **Datasets:**

For SVM, we chose to run this algorithm on two datasets: (1) the Iris dataset used in hw3 (2)Breast Cancer Wisconsin (Diagnostic) dataset

- **Hard Margin and Soft Margin SVMs using Sklearn**

class `sklearn.svm.` **SVC**(*C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape='ovr', break_ties=False, random_state=None*) [source]

In Sklearn's SVM, we can see it's using C as regularization parameter that controls how much you want to punish your model for each misclassified point for a given curve. The lower the C parameters, the softer the margin.
Since we used linear kernel for SVM during the class, we also chose Linear Kernel for the Irsi dataset. Hoewever, to explore soft margin's effect, we also used RBF kernel on the other dataset (Breast Cancer db)
For hard margin SVM, using linear Kernel we set param C to 1000 and for soft margin SV, we set param C to 1.5 for both data sets mentioned above.

- **Soft Margin SVM built using Numpy**

We also used C =1.5 and Linear Kernel when we implemented our own Soft margin extension on SVM as shown below:

Get w, w_0 and α when $C = 1.5$ using the below function

```
: w,b,alphas = soft_dual_svm(X_train,y_train ,1.5)
  # print('alphas = ',alphas)
  print('w = ', w.flatten())
  print('b = ', b[0])
```

```
w =  [ 1.1725151   4.02254107 -4.24359542 -7.15215218]
b =  [14.45316144]
```

The **decision function for dual problem** is $f_{\text{dual}}(\mathbf{x}) = \left(\sum_{i \in I} \alpha^{(i)} y^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x})\right) + w_0$ where $I = \{i \mid \alpha^{(i)} \neq 0\}$.
In this extension, we will use a linear kernel function.

2.3 Accuracies Table for SVM

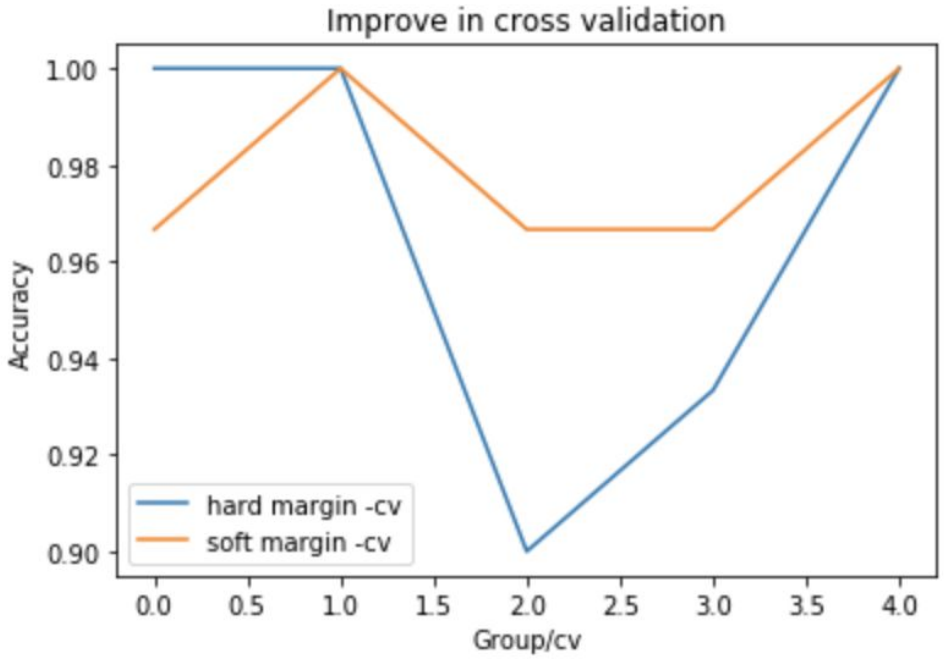
Note: since both Iris dataset and Breast cancer dataset are relatively small data set, we used cross-validation function from Sklearn for calculating the accuracies

Algo	Dataset	Param + Kernel	Accuracy
Sklearn Hard margin SVM	Iris	C = 1000, Linear	0.97 (+/- 0.08) (cross-val)
Sklearn Soft margin SVM	Iris	C = 1.5, Linear	0.98 (+/- 0.03) (cross-val)
Sklearn Hard margin SVM	Breast Cancer	C = 1000, RBF	0.91 (+/- 0.04) (cross-val)
Sklearn Soft margin SVM	Breast Cancer	C = 1.5, RBF	0.92 (+/- 0.03) (cross-val)
Numpy Soft margin SVM	Iris	C = 1.5, Linear	0.878 (test val)
Numpy Soft margin SVM	Breast Cancer	C = 1.5, Linear	0.947 (test val)

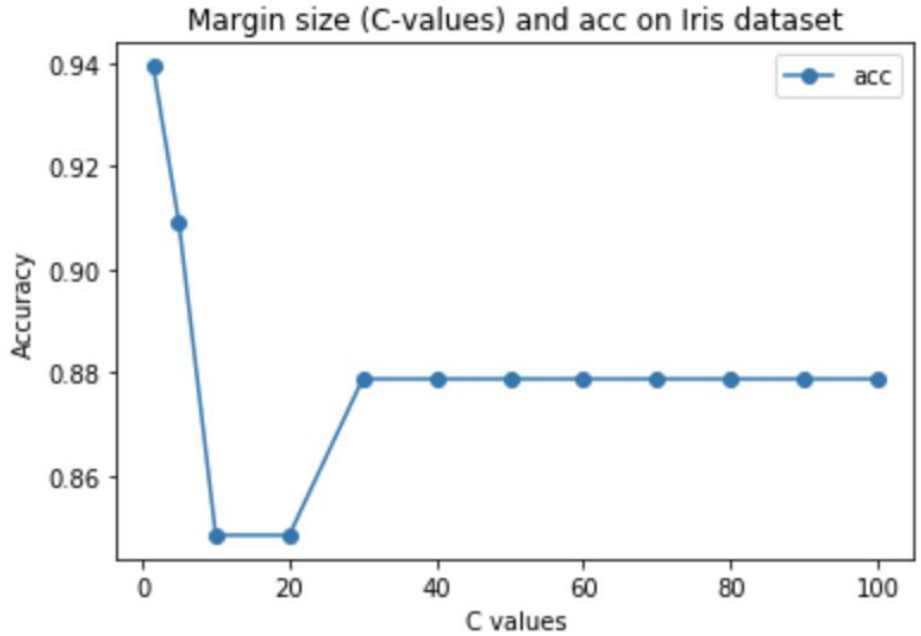
2.4 Explanations on specific datasets for SVM

- **Iris Data**

Using Sklearn, we can see some improvement that for each cross validation set:

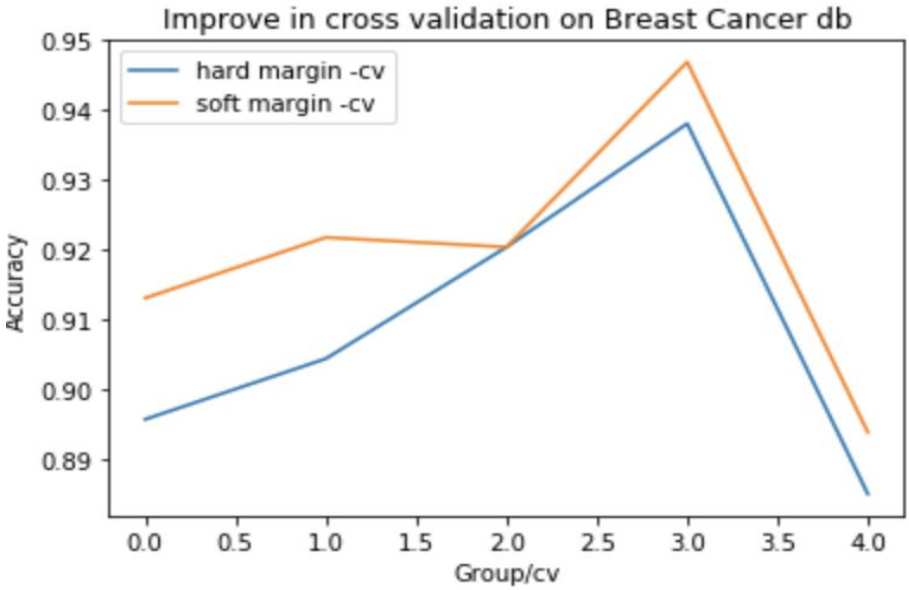


In our own **Numpy implementation**, we analyzed how accuracy might improve on Iris using different param C values:

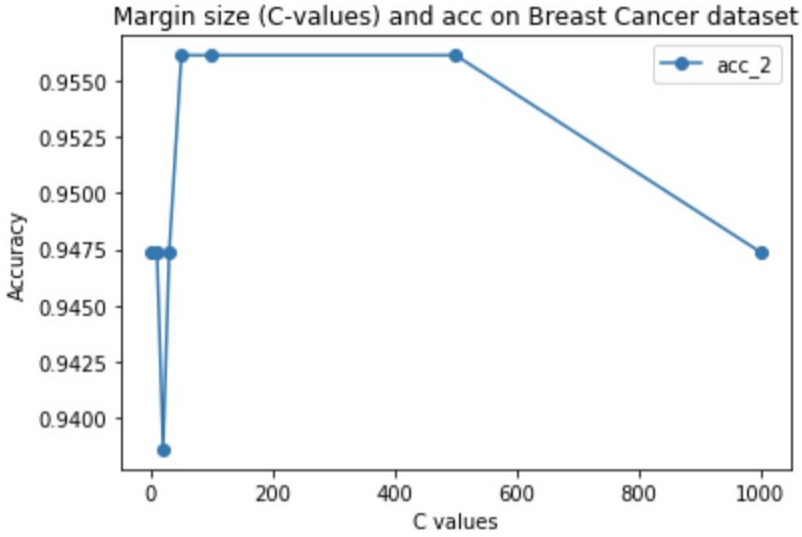


- **Breast Cancer Dataset**

Using Sklearn, we can see some improvement that for each cross validation set:



In our own **Numpy implementation**, we analyzed how accuracy might improve on Brest Cancer Dataset using different param C values:



From the chart above, different from the Iris data set, we can see that the accuracy first goes up and then goes down as we increase the C value. One possible explanation is at the beginning we have very small penalty so that there's an under fitting problem

2.5 Numpy Code for soft magin SVM

```
# This is the dual problem solution for soft margin SVM
def soft_dual_svm(X,y,C):
    solvers.options['show_progress'] = False
    NUM,DIM = X.shape
    y = y.reshape(-1,1) * 1.
    temp = y * X
    H = np.dot(temp , temp.T) * 1.
#   Match vectors to the CVXOPT's matrix format:
    P = matrix(H)
    q = matrix(-np.ones((NUM, 1)))
    G = matrix(np.vstack((np.eye(NUM)*-1,np.eye(NUM))))
    h = matrix(np.hstack((np.zeros(NUM), np.ones(NUM) * C)))
    A = matrix(y.reshape(1, -1))
    b = matrix(np.zeros(1))
#   Use CVXOPT's quadratic peoblem solver -> the returned solutions are the values of all alphas
    result = solvers.qp(P, q, G, h, A, b)
    alphas = np.array(result['x'])
    w = ((y * alphas).T @ X).reshape(-1,1)
    alpha = (alphas > 1e-3).flatten()
    b = y[alpha] - np.dot(X[alpha], w)
    return w,b,alphas
```

The **decision function for dual problem** is $f_{\text{dual}}(\mathbf{x}) = \left(\sum_{i \in I} \alpha^{(i)} y^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x})\right) + w_0$ where $I = \{i \mid \alpha^{(i)} \neq 0\}$. In this extension, we will use a linear kernel function.

```
def lin_kernel(xi, xj):
    return np.dot(xi,xj)
```

```
def f_dual(xx,alphas,b,X_train,y_train):
    sum = 0
    for i in range(X_train.shape[0]):
        sum+=y_train[i]*alphas[i]*lin_kernel(X_train[i].transpose(),xx)
    sum+=b[0]
    result = sum
#     print(result)
    if result >0:
        return 1
    else:
        return -1
    return result
```

```
def predict(X_test,y_test,alphas,b,X_train,y_train):
    predicted_y = np.zeros(len(X_test))
    for i in range(len(X_test)):
        predicted_y[i]=f_dual(X_test[i],alphas,b,X_train,y_train)
    correct = 0
    for j in range(len(y_test)):
        if predicted_y[j] == y_test[j]:
            correct += 1
    acc = float(correct/len(y_test))
#     print("Correctly predicted test data number is ", correct)
#     print("Accuracy on test data is ", acc)
    return acc
```

3. Lasso Regression with Coordinate descent

3.1 Explanation about Lasso with coordinate descent

In class, we implemented the Ridge Regression with L2 regularization. In the lecture, the professor also covered Lasso regression with L1 regularization. Not like Ridge Regression that has a closed form solution, the **Lasso Regression has no closed form** solution to minimize :

$$E_{lasso}(w) = E_{in}(w) + \lambda||w||_1$$

Although *Elasso* (*w*) is convex, we can not take the derivative and set it to zero to find the optimal *w*. However, it's possible to optimize the *jth* coefficient while the others remain fixed:

$$w_j^* = argmin_z E_{lasso}(w + ze_j)$$

One way to solve optimization problems is coordinate descent, in which at each step we optimize over one component of the unknown parameter vector, fixing all other components. The descent path so obtained is a sequence of steps, each of which is parallel to a coordinate axis in Rd, hence the name. It turns out that for the Lasso optimization problem, we can find a closed form solution for optimization over a single component fixing all other components.
In the Numpy code, we will implement the coordinated Descent Algorithm (taken from class slides "Lec 04 Regularization" page 23) :

- Initialize $w = (X^T X + \lambda I)^{-1} X^T y$
- To decrease the cost, update the parameters one at a time:
  - for j = i to d:
  - $a_j = \frac{2}{N} \sum_{i=1}^{N} (x_j^{(i)})^2$
  - $c_j = \frac{2}{N} \sum_{i=1}^{N} x_j^{(i)} (y^{(i)} - W_j^T x_j)^2$
  - $w_j = \frac{(c_j + \lambda)}{a_j}$ if $c_j < -\lambda$
  - $w_j = 0$ if c_j is in $[-\lambda, \lambda]$
  - $w_j = \frac{(c_j - \lambda)}{a_j}$ if $c_j > -\lambda$

3.2 Summary of Lasso Regression's Implementation

- **Dataset**

In this extension, we decided to use the **Boston Housing** (Dataset used in HW4) and **NYSE Stock market prices** dataset (from outside class) [Data Source: https://www.kaggle.com/dgawlik/nyse/kernels]

- **Sklearn implementation of Linear, Ridge and Lasso regressions**

Since in homework 4 we implemented the ridge regression, we think it's interesting to see how pure linear regression, ridge regression and lasso regression works on the same dataset.

```
from sklearn.linear_model import LinearRegression
linear = LinearRegression()
linear.fit(X_train,Y_train)
#predictions on training data
Y_pred_train_linear = linear.predict(X_train)
Y_pred_test_linear = linear.predict(X_test)
```

Remember that we did Ridge regression in class with k-fold validation and found the best alpha for the housing price dataset which is alpha = 1.49, so we will use this alpha when implementing Ridge Regression from Sklearn here.

```
from sklearn.linear_model import Ridge
rr = Ridge(alpha=1.49)
rr.fit(X_train,Y_train)
 #predictions on training data
Y_pred_train_rr = rr.predict(X_train)
#predictions on testing data
Y_pred_rr = rr.predict(X_test)
```

And for Lasso, we used alpha = 0.03 in Sklearn implementation.

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.03)
lasso.fit(X_train,Y_train)
#predictions on training data
print(lasso.coef_)
Y_pred_train_lasso = lasso.predict(X_train)
Y_pred_test_lasso= lasso.predict(X_test)
```

- **Numpy Implementation**

We set alpha to 0.05 and the max number of iteration for coordinate descent for 1,000

```
partia_min_theta(x,
lambd = 0.05
n = len(y)
```

```
# initial_theta = np.zeros((13,1))
beta_init = np.zeros(np.size(X_train, 1))
theta = coorddescent(X_train,Y_train,beta_init,1000)
```

```
Coordinate descent iteration 100
Coordinate descent iteration 200
Coordinate descent iteration 300
Coordinate descent iteration 400
Coordinate descent iteration 500
Coordinate descent iteration 600
Coordinate descent iteration 700
Coordinate descent iteration 800
Coordinate descent iteration 900
Coordinate descent iteration 1000
```

3.3 Accuracies Table for Lasso

Note for regression problem, we used Mean Squared Error (MSE), Mean Absolute Error(MAE), R-square scores(R2) to measure accuracies:

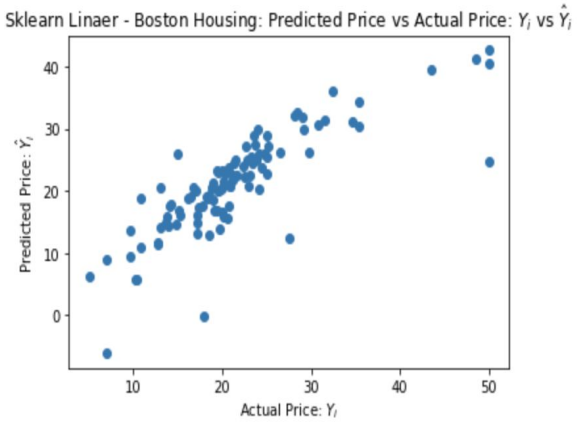
Algo	Dataset	Parameters	MSE	R2	MAE
Sklearn-Linear	Boston Housing	N/A	24.29	0.6687	3.18
Sklearn-Ridge	Boston Housing	Alpha = 1.49	24.5	0.665	3.128
Sklearn-Lasso	Boston Housing	Alpha = 0.03	24.5	0.6689	3.12
Numpy Lasso	Boston Housing	Apha = 0.05	25.77	0.6467	3.428
Sklearn-Linear	NYSE Stock	N/A	3550.00	0.856	22.22
Sklearn-Ridge	NYSE Stock	Alpha = 1	1371.82	0.943	17.77
Sklearn-Lasso	NYSE Stock	Alpha = 0.03	1362.9	0.94	17.77
Numpy Lasso	NYSE Stock	Apha = 0.05	1754.63	0.941	19.78

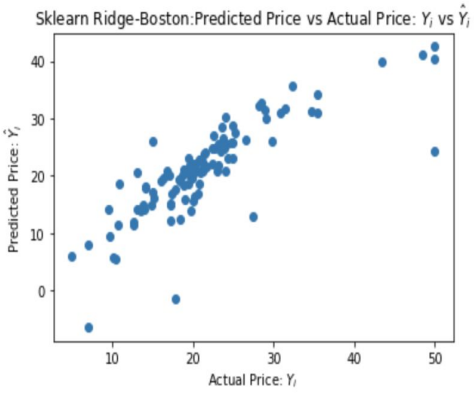
One reason for why Lasso on GOOGLE Stock price's MSE is very high is because the y values which are the stock prices values are very large in themselves and the data has some outliers. Over all, since the R2 scores are pretty high, we can see that our models fits the dataset pretty well.

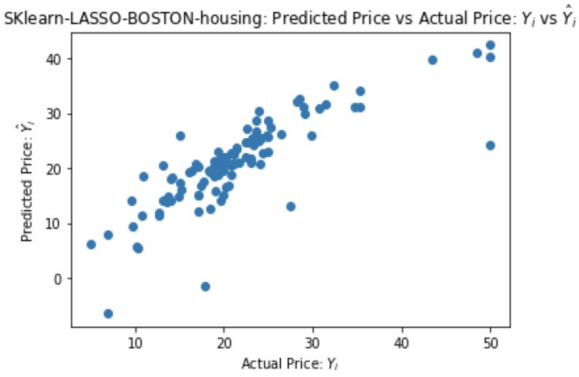
3.4 Explanations on Specific Datasets Lasso

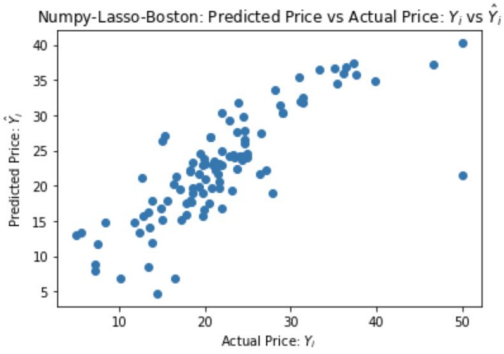
Let's see how well our predictions of y compared to the real label of y for each Dataset:

- **Boston housing:**



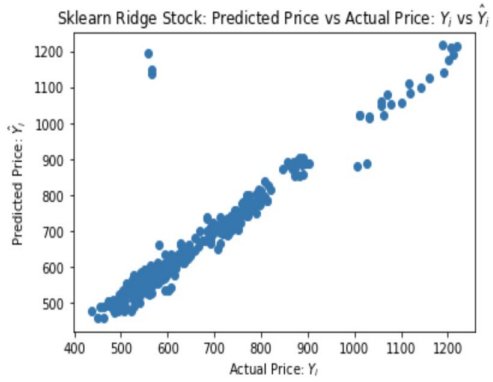


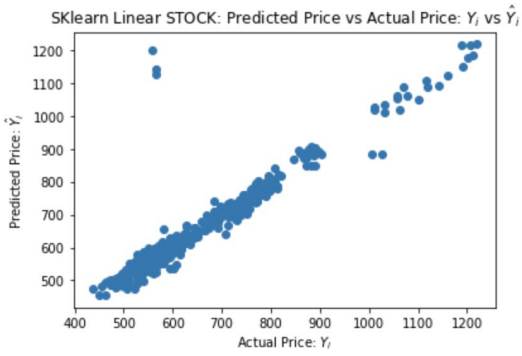


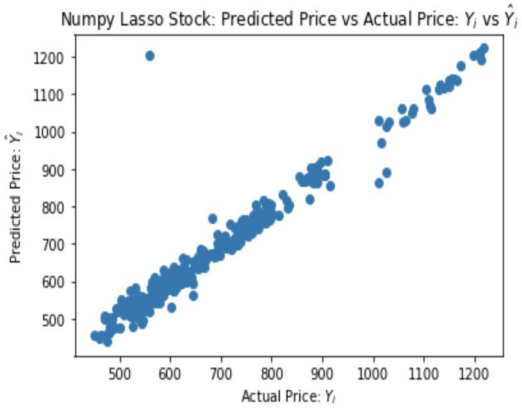


We can see here both three regression from SKlearn has pretty similar performance, one reason might be we don't have many outliers here in the Boston housing dataset and the Sklearn's linear regression doesn't overfit or under fit, it's already very good.

- **NYSE Stock prices (GOOGLE)**







As we can see here and the accuracies table above, Lasso and Ridge perform better than Linear on NYSE stocks market both Sklearn implementation and our own Numpy implementation. This might because the original dataset has some terrible outliers so the Linear model might over-fit.

3.5 Numpy code for Lasso

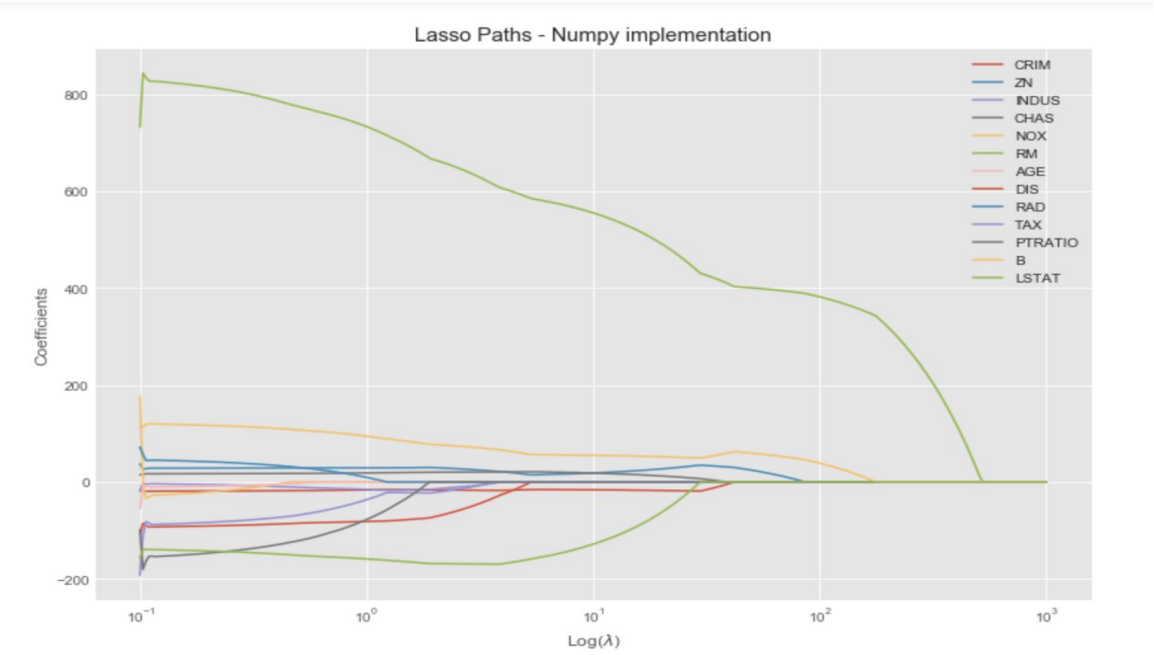
```
def soft_threshold(x, lambd):
#     This is the soft_threshold function, which takes in c_j and lambda values, return the corresponding w_j values
    if x < -lambd:
        return x+lambd
    elif x > lambd:
        return x-lambd
    else:
        return 0
```

```
# This solves the partial minimization problem with respect to theta_j for any j = 1...d.
def partia_min_theta(x, y, theta, j):
    lambd = 0.05
    n = len(y)
    selector = [i for i in range(x.shape[1]) if i != j]
    norm_x_j = np.linalg.norm(x[:, j])
    a = x[:, j].dot(y[:, np.newaxis] - x[:, selector].dot(theta[:, np.newaxis][selector, :]))
    passin = lambd*n/2
#     print(passin)
    res = soft_threshold(a, passin)
    return res/(norm_x_j**2)
```

```
def coorddescent(x, y, theta_init,max_iter):
#     This is our coordiante descent function, stops at max iteration.
    theta = copy.deepcopy(theta_init)
    theta_vals = theta
    d = np.size(x, 1)
    iter = 0
    while iter < max_iter:
        for j in range(d):
            min_theta_j = partia_min_theta(x, y, theta, j)
            theta[j] = min_theta_j
        theta_vals = np.vstack((theta_vals, theta))
        iter += 1
        if iter % 100 == 0:
            print('Coordinate descent iteration', iter)
    return theta_vals
```

3.6 Further explanation on Lasso regression

Although for most datasets that Lasso regression and Ridge Regression can achieve similar MSEs and R2 scores in terms of regularization.
There's one feature that only Lasso regression has can be very useful -- **feature selection**.
This is because the coefficient of Lasso regression path can converge to 0 while it never happens in Ridge regression.
For example, the chart below showed the Lasso Path on Boston data:



4 Neural Networks with Convolutional layers (CNN)

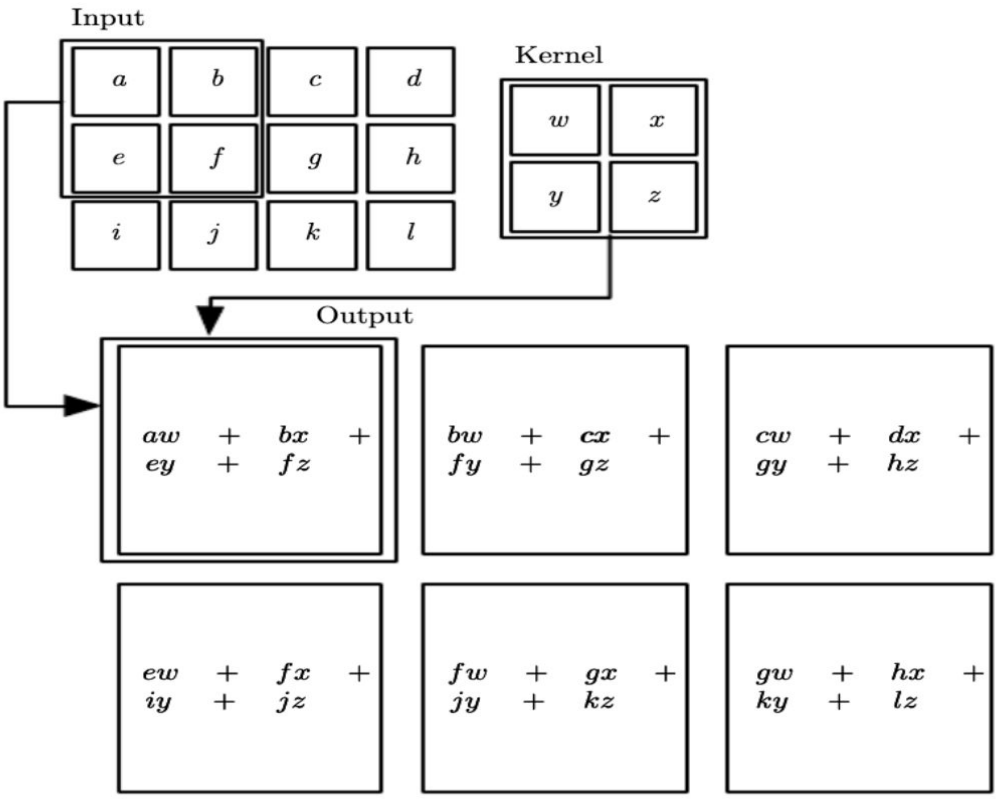
4.1 Explanation about CNN Algorithm

The key operation performed in CNN layers is that of 2D convolution. In homework8, we implemented neural networks with fully connected layers.

Difference between fully connected layers and convolutional layers:

In a fully connected layer, each neuron receives input from every element of the previous layer. In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer.

In CNNs, each member of the kernel is used at every feasible position of the input. The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.



4.2 Summary of CNN implementation

(1) 2 Convolution layer which has several filters
(2) The weights are initialized randomly
(3) Using ReLU as activation function
(4) Using MaxPool Method when passing the layer
(5) Flatten the layer at the end with softmax
(6) When training, using gradient descent as well
(7) Using forward and backpropagation to update parameters

4.3 Accuracies Table

	ACC for Dataset in homework (3000 samples)	ACC for Minst Data (1797 samples)	Number of layers	Activation Function	Learning Rate
NN in homework	86.5	75.6	2	sigmoid	0.25

CNN with Pytorch	88.5	88.0	2	Relu	0.07
CNN with Numpy	93.5	94.0	2	Relu	0.02

4.4 Specific explanation on datasets

The dataset using is Minst handwritten digit database http://yann.lecun.com/exdb/mnist/, which contains images of handwritten digit from 0 -9.
Since it takes a long time to train, there's only one final result. The accuracy of CNN varies a lot with different parameters. For example, the accuracy of Numpy implement would drop to 83.6% for the dataset in homework. There might be some difference from the gradient descent function of Pytorch and numpy implementation so that they didn't have the same performance with the same hyper parameters. We try to find the best hyper parameters for both ones, so we choose different alpha as well as the params of layers. The reason why numpy implementation performs better that Pytorch is perhaps because of differences in implementation and Pytorch didn't have optimal hyper parameters.
CNN performs better than NN because it uses the maxpool method to filter the most significant digit with a moving window, and it can extract the shape from the figure by convolution if there's an obvious shape. This is useful to identify features from the figure.

4.5 Numpy Code

```
def conv(image, label, params, conv_s, pool_f, pool_s):

    [f1, f2, w1, w2, b1, b2, b3, b4] = params
    #w1 &b3 are layer1, w2 &b4 are layer 2, f1&b1 are convolution of layer1, f2&b2 are  convoluti
on of layer2

    #feedforward
    conv1 = convolution(image, f1, b1, conv_s)
    conv1[conv1<=0] = 0 #use Relu instead of f(z)
    #conv1 = f(conv1)

    conv2 = convolution(conv1, f2, b2, conv_s)
    conv2[conv2<=0] = 0
    #conv2 = f(conv2)

    pooled = maxpool(conv2, pool_f, pool_s) # find the max in neighbours

    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1))
    z = w1.dot(fc) + b3
    #z = f(z)
    z[z<=0] = 0

    out = w2.dot(z) + b4

    probs = softmax(out)
    loss = categoricalCrossEntropy(probs, label)
```

```
    #backpropagation

    dout = probs - label
    dw2 = dout.dot(z.T)
    db4 = np.sum(dout, axis = 1).reshape(b4.shape)

    dz = w2.T.dot(dout)
    #dz = f_deriv(dz)
    dz[z<=0] = 0 # backpropagate through ReLU
    dw1 = dz.dot(fc.T)
    db3 = np.sum(dz, axis = 1).reshape(b3.shape)

    dfc = w1.T.dot(dz)
    dpool = dfc.reshape(pooled.shape) # reshape fully connected into dimensions of pooling layer

    dconv2 = maxpoolBackward(dpool, conv2, pool_f, pool_s)
    dconv2[conv2<=0] = 0 # backpropagate through ReLU
    #dconv2 = f_deriv(dconv2)

    dconv1, df2, db2 = convolutionBackward(dconv2, conv1, f2, conv_s)
    dconv1[conv1<=0] = 0 # backpropagate through ReLU
    #dconv1 = f_deriv(dconv1)

    dimage, df1, db1 = convolutionBackward(dconv1, image, f1, conv_s)

    grads = [df1, df2, dw1, dw2, db1, db2, db3, db4]

    return grads, loss
```

```
def initializeFilter(size, scale = 1.0):
    stddev = scale/np.sqrt(np.prod(size))
    return np.random.normal(loc = 0, scale = stddev, size = size)

def initializeWeight(size):
    return np.random.standard_normal(size=size) * 0.005
```

```
def graduate_descent(batch, num_classes, alpha, dim, n_c, beta1, beta2, params, cost):

    [f1, f2, w1, w2, b1, b2, b3, b4] = params

    X = batch[:,0:-1] # get batch inputs
    X = X.reshape(len(batch), n_c, dim, dim)
    Y = batch[:,-1] # get batch labels

    cost_ = 0
    batch_size = len(batch)

    # initialize gradients and params
    df1 = np.zeros(f1.shape)
    df2 = np.zeros(f2.shape)
    dw1 = np.zeros(w1.shape)
    dw2 = np.zeros(w2.shape)
    db1 = np.zeros(b1.shape)
    db2 = np.zeros(b2.shape)
    db3 = np.zeros(b3.shape)
    db4 = np.zeros(b4.shape)

    v1 = np.zeros(f1.shape)
    v2 = np.zeros(f2.shape)
    v3 = np.zeros(w1.shape)
    v4 = np.zeros(w2.shape)
    bv1 = np.zeros(b1.shape)
    bv2 = np.zeros(b2.shape)
    bv3 = np.zeros(b3.shape)
    bv4 = np.zeros(b4.shape)

    s1 = np.zeros(f1.shape)
    s2 = np.zeros(f2.shape)
    s3 = np.zeros(w1.shape)
    s4 = np.zeros(w2.shape)
    bs1 = np.zeros(b1.shape)
```

```
    bs2 = np.zeros(b2.shape)
    bs3 = np.zeros(b3.shape)
    bs4 = np.zeros(b4.shape)

    for i in range(batch_size):

        x = X[i]
        y = np.eye(num_classes)[int(Y[i])].reshape(num_classes, 1) # convert label to one-hot

        # Collect Gradients for training example
        grads, loss = conv(x, y, params, 1, 2, 2)
        [df1_, df2_, dw1_, dw2_, db1_, db2_, db3_, db4_] = grads

        df1+=df1_
        db1+=db1_
        df2+=df2_
        db2+=db2_
        dw1+=dw1_
        db3+=db3_
        dw2+=dw2_
        db4+=db4_

        cost_+= loss

    # Parameter Update

    v1 = beta1*v1 + (1-beta1)*df1/batch_size # momentum update
    s1 = beta2*s1 + (1-beta2)*(df1/batch_size)**2 # RMSProp update
    f1 -= alpha * v1/np.sqrt(s1+1e-7) # combine momentum and RMSProp to perform update with Adam

    bv1 = beta1*bv1 + (1-beta1)*db1/batch_size
    bs1 = beta2*bs1 + (1-beta2)*(db1/batch_size)**2
    b1 -= alpha * bv1/np.sqrt(bs1+1e-7)
```

```
    v2 = beta1*v2 + (1-beta1)*df2/batch_size
    s2 = beta2*s2 + (1-beta2)*(df2/batch_size)**2
    f2 -= alpha * v2/np.sqrt(s2+1e-7)

    bv2 = beta1*bv2 + (1-beta1) * db2/batch_size
    bs2 = beta2*bs2 + (1-beta2)*(db2/batch_size)**2
    b2 -= alpha * bv2/np.sqrt(bs2+1e-7)

    v3 = beta1*v3 + (1-beta1) * dw1/batch_size
    s3 = beta2*s3 + (1-beta2)*(dw1/batch_size)**2
    w1 -= alpha * v3/np.sqrt(s3+1e-7)

    bv3 = beta1*bv3 + (1-beta1) * db3/batch_size
    bs3 = beta2*bs3 + (1-beta2)*(db3/batch_size)**2
    b3 -= alpha * bv3/np.sqrt(bs3+1e-7)

    v4 = beta1*v4 + (1-beta1) * dw2/batch_size
    s4 = beta2*s4 + (1-beta2)*(dw2/batch_size)**2
    w2 -= alpha * v4 / np.sqrt(s4+1e-7)

    bv4 = beta1*bv4 + (1-beta1)*db4/batch_size
    bs4 = beta2*bs4 + (1-beta2)*(db4/batch_size)**2
    b4 -= alpha * bv4 / np.sqrt(bs4+1e-7)


    cost_ = cost_/batch_size
    cost.append(cost_)

    params = [f1, f2, w1, w2, b1, b2, b3, b4]

    return params, cost
```

```
: def train(X,y,num_classes = 10, alpha = 0.01, beta1 = 0.95, beta2 = 0.99, img_dim = 8, img_depth = 1, f = 5, num_filt1
    # after you change the img_dim, you should change f as well as w3(line208 to make sure two matrix can dot)
    # training data

    #m =50
    #digits = load_digits()
    #X = digits.data
    #X = extract_data('train-images-idx3-ubyte.gz', m, img_dim)


    #y_dash = np.array([[yy] for yy in digits.target])
    #y_dash = extract_labels('train-labels-idx1-ubyte.gz', m).reshape(m,1)
    train_data = np.hstack((X,y))

    np.random.shuffle(train_data)

    ## Initializing all the parameters
    if img_dim ==28:
        f = 5
        num_filt1 = 8
        num_filt2 = 8
        f1, f2, w1, w2 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (128,800), (10, 128)

    elif img_dim == 8:
        f = 2
        f1, f2, w1, w2 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f), (128,288), (10, 128)
    else:
        print("Thia img_dim has not been implemented")
    f1 = initializeFilter(f1)
    f2 = initializeFilter(f2)
    w1,b3 = setup_and_init_weights_cnn(w1)
    w2,b4 = setup_and_init_weights_cnn(w2)
    b1 = np.zeros((f1.shape[0],1))
    b2 = np.zeros((f2.shape[0],1))
```

```
    params = [f1, f2, w1, w2, b1, b2, b3, b4]

    cost = []

    print("LR:"+str(alpha)+", Batch Size:"+str(batch_size))

    for epoch in range(num_epochs):
        np.random.shuffle(train_data)
        batches = [train_data[k:k + batch_size] for k in range(0, train_data.shape[0], batch_size)]
        for batch in batches:
            params, cost = graduate_descent(batch, num_classes, alpha, img_dim, img_depth, beta1, beta2, params, cost)

    return (params, cost)
```

```
def convolution(image, filt, bias, s=1):
    '''
    Confolves `filt` over `image` using stride `s`
    '''
    (n_f, n_c_f, f, _) = filt.shape # filter dimensions
    n_c, in_dim, _ = image.shape # image dimensions

    out_dim = int((in_dim - f)/s)+1 # calculate output dimensions


    out = np.zeros((n_f,out_dim,out_dim))

    # convolve the filter over every part of the image, adding the bias at each step.
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= in_dim:
            curr_x = out_x = 0
            while curr_x + f <= in_dim:
                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] * image[:,curr_y:curr_y+f, curr_x:curr_x+f]) + bias[cur
                curr_x += s
                out_x += 1
```

```
    return out
```

```
def maxpool(image, f=2, s=2):
    '''
    Downsample `image` using kernel size `f` and stride `s`
    '''
    n_c, h_prev, w_prev = image.shape

    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1

    downsampled = np.zeros((n_c, h, w))
    for i in range(n_c):
        # slide maxpool window over each part of the image and assign the max value at each step to the output
        curr_y = out_y = 0
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            while curr_x + f <= w_prev:
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled
def softmax(X):
    out = np.exp(X)
    return out/np.sum(out)
def categoricalCrossEntropy(probs, label):
    return -np.sum(label * np.log(probs))
```

```
def convolutionBackward(dconv_prev, conv_in, filt, s):
    '''
    Backpropagation through a convolutional layer.
    '''
    (n_f, n_c, f, _) = filt.shape
    (_, orig_dim, _) = conv_in.shape
    ## initialize derivatives
    dout = np.zeros(conv_in.shape)
    dfilt = np.zeros(filt.shape)
    dbias = np.zeros((n_f,1))
    for curr_f in range(n_f):
        # loop through all filters
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                # loss gradient of filter (used to update the filter)
                dfilt[curr_f] += dconv_prev[curr_f, out_y, out_x] * conv_in[:, curr_y:curr_y+f, curr_x:curr_x+f]
                # loss gradient of the input to the convolution operation (conv1 in the case of this network)
                dout[:, curr_y:curr_y+f, curr_x:curr_x+f] += dconv_prev[curr_f, out_y, out_x] * filt[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
        # loss gradient of the bias
        dbias[curr_f] = np.sum(dconv_prev[curr_f])

    return dout, dfilt, dbias
```

```python
def maxpoolBackward(dpool, orig, f, s):
    '''
    Backpropagation through a maxpooling layer. The gradients are passed through the indices of greatest value in th
    '''
    (n_c, orig_dim, _) = orig.shape

    dout = np.zeros(orig.shape)

    for curr_c in range(n_c):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                # obtain index of largest value in input for current window
                (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f, curr_x:curr_x+f])
                dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]

                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return dout
```

```
digits = load_digits()
X = digits.data
y = np.array([[yy] for yy in digits.target])
X_scale = StandardScaler()
X = X_scale.fit_transform(digits.data)
X_train, X_test, y_train, y_test = train_test_split(digits.data, y, test_size=0.4)
img_dim = 8
(params, cost) = train(X = X_train, y = y_train,alpha = 0.02, img_dim = img_dim)
```

```
LR:0.02, Batch Size:32
```

```
[f1, f2, w1, w2, b1, b2, b3, b4] = params
```

```
def predict(image, f1, f2, w1, w2, b1, b2, b3, b4, conv_s = 1, pool_f = 2, pool_s = 2):
    '''
    Make predictions with trained filters/weights.
    '''
    conv1 = convolution(image, f1, b1, conv_s) # convolution operation
    conv1[conv1<=0] = 0 #relu activation

    conv2 = convolution(conv1, f2, b2, conv_s) # second convolution operation
    conv2[conv2<=0] = 0 # pass through ReLU non-linearity

    pooled = maxpool(conv2, pool_f, pool_s) # maxpooling operation
    (nf2, dim2, _) = pooled.shape
    fc = pooled.reshape((nf2 * dim2 * dim2, 1)) # flatten pooled layer

    z = w1.dot(fc) + b3 # first dense layer
    z[z<=0] = 0 # pass through ReLU non-linearity

    out = w2.dot(z) + b4 # second dense layer
    probs = softmax(out) # predict class probabilities with the softmax activation function

    return np.argmax(probs), np.max(probs)
```

```
X = X_test
y_dash = y_test
X-= int(np.mean(X)) # subtract mean
X/= int(np.std(X)) # divide by standard deviation
test_data = np.hstack((X,y_dash))

X = test_data[:,0:-1]
X = X.reshape(len(test_data), 1,img_dim, img_dim)
y = test_data[:,-1]

corr = 0
digit_count = [0 for i in range(10)]
digit_correct = [0 for i in range(10)]

#t = tqdm(range(len(X)), leave=True)

for i in range(len(X)):
    x = X[i]
    pred, prob = predict(x, f1, f2, w1, w2, b1, b2, b3, b4)
    digit_count[int(y[i])]+=1
    if pred==y[i]:
        corr+=1
        digit_correct[pred]+=1

   # t.set_description("Acc:%0.2f%%" % (float(corr/(i+1))*100))

print("ACC: "+ str(float(corr/len(test_data)*100)))
```

```
ACC: 93.46314325452016
```