CS 189: Introduction to Machine Learning

Fall 2017

Homework 13

Solutions by

Jinhong Du

3033483677

Question 1

(a)

Jinhong Du
jaydu@berkeley.edu

(b)

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up. Jinhong Du

Question 2

(a)

$\because$

$$
\begin{aligned}
\sigma(t) &= \frac{1}{1+e^{-t}} \\
\lim_{t\to+\infty} \sigma(t) &= \frac{1}{1+\lim\limits_{t\to+\infty} e^{-t}} \\
&= 1 \\
\lim_{t\to-\infty} \sigma(t) &= \frac{1}{1+\lim\limits_{t\to-\infty} e^{-t}} \\
&= 0
\end{aligned}
$$

and $\sigma(t)$ is bounded by 1

$\therefore \quad \sigma(t)$ is a thresholding function

$\because$

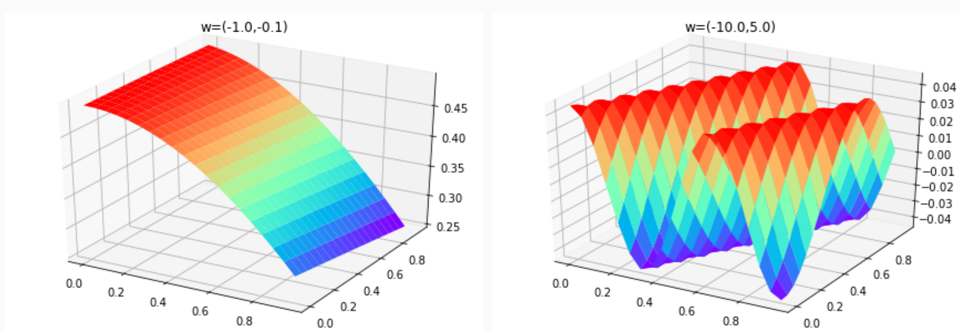
$$
\begin{aligned}
f(t) &= Relu(t) - Relu(t-1) \\
&= \begin{cases} 1 & ,t \geqslant 1 \\ t & ,0 < t < 1 \\ 0 & ,t \leqslant 0 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\lim_{t\to+\infty} f(t) &= 1 \\
\lim_{t\to-\infty} f(t) &= 0
\end{aligned}
$$

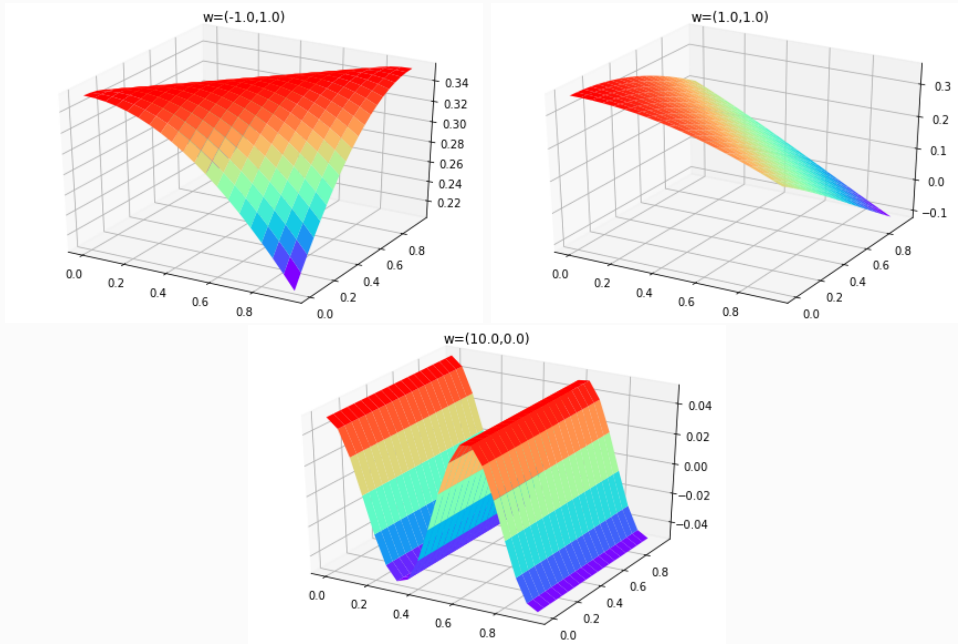
and $f(t)$ is bounded by 1

$\therefore \quad Relu(t) - Relu(t-1)$ is a thresholding function

(b)



Solution (cont.)



(c)

Let

$$\tau_n(t) = \tau(nt)$$

then $\forall\ t_0 > 0$,

$$\begin{aligned}\lim_{n\to+\infty} \tau_n(t_0) &= \lim_{n\to+\infty} \tau(nt_0)\\ &= 1\end{aligned}$$

$\forall\ t_0 < 0$,

$$\begin{aligned}\lim_{n\to+\infty} \tau_n(t_0) &= \lim_{n\to+\infty} \tau(nt_0)\\ &= 0\end{aligned}$$

$\therefore$

$$\lim_{n\to\infty} \tau_n(t) = \begin{cases} 1 & , t > 0 \\ \tau(0) & , t = 0 \\ 0 & , t < 0 \end{cases}$$

For $t_0 = 0$, $\because$

$$\lim_{t\to\infty} \tau(nt + n) = 1$$

$\therefore$

$$S(t) \subseteq cl(\{\tau(nt) \text{ for all } n \in \mathbb{N}\})$$

i.e. threshold functions with appropriately scaled arguments are step functions in the limit.

$\because \quad \forall\ (w', b') \in \mathbb{R}^d \times \mathbb{R}$, $x \in \mathbb{R}^d$,

$$S(< w', x > +b') = \begin{cases} 1 & , < w', x > +b' \geqslant 0 \\ 0 & , < w', x > +b' < 0 \end{cases}$$

Solution (cont.)

is a step functions

$\therefore$

$$
\begin{aligned}
S(<w', x> + b') &\in cl(\{\tau[n(<w', x> + b')] \text{ for all } n \in \mathbb{N}\}) \\
&\subseteq cl(\{\tau(<w, x> + b) \text{ for some } w, b\})
\end{aligned}
$$

(d)

$\because$ $c(y')$ is a continous function on $[-1, 1]$

$\therefore$ $c(y')$ is uniformly continuous on $[-1, 1]$, i.e., $\forall\ \delta > 0$, $\exists \epsilon > 0$, s.t. $\forall\ x_1, x_2 \in [-1, 1]$, $|x_1 - x_2| < \epsilon$,

$$
|c(x_1) - c(x_2)| < \delta
$$

$\because$ $\forall\ a, b \in \mathbb{R}, a < b$,

$$
\mathbb{1}_{[a,b]}(t) = S(t - a) - S(t - b)
$$

$\therefore$ $\exists\ n \in \mathbb{N}^+$, s.t. $\Delta : -1 = x_0 < x_1 < \cdots < x_n = 1$, $x_n - x_{n-1} < \epsilon$ and $\forall\ i \in \mathbb{N}^+, i \leqslant n,\ y' \in [x_{i-1}, x_i]$,

$$
\begin{aligned}
g(y') &= \min_{t \in [x_{i-1}, x_i]} \{c(t)\} \cdot \mathbb{1}_{[x_{i-1}, x_i]}(y') \\
&= \min_{t \in [x_{i-1}, x_i]} \{c(t)\} \cdot [S(y' - x_{i-1}) - S(y' - x_i)]
\end{aligned}
$$

$$
|g(y') - c(y')| < \delta
$$

Therefore, $\forall\ y' \in [-1, 1]$,

$$
|g(y') - c(y')| < \delta
$$

Let

$$
c(y_i) = \min_{t \in [x_{i-1}, x_i]} \{c(t)\}
$$

then

$$
\begin{aligned}
g(y') &= \sum_{i=1}^{n} c(y_i)[S(y' - x_{i-1}) - S(y' - x_i)] \\
&= \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} [c(y_i) - c(y_{i-1})]S(y' - x_{i-1})
\end{aligned}
$$

and $P_\delta = \{y_1, \cdots, y_n\}$

(e)

$$
\begin{aligned}
\sum_i |c(z_i) - c(z_{i-1})| &\leqslant \lim_{\delta \to 0} \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} \frac{|\cos(\|w\|_1 y_i) - \cos(\|w\|_1 y_{i-1})|}{(y_i - y_{i-1})} (y_i - y_{i-1}) \\
&\leqslant \int_{-1}^{1} |c(x)|' \mathrm{d}x \\
&= \|w\|_1 \int_{-1}^{1} |\sin(\|w\|_1 x)| \mathrm{d}x \\
&\leqslant \|w\|_1 \int_{-1}^{1} 1 \mathrm{d}x \\
&\leqslant 2\|w\|_1
\end{aligned}
$$

where $\delta = \max\{x_i - x_{i-1} : 0 < 1 \leqslant n\}$

Therefore, for every $w \neq 0$, we have approximated $\cos(< w, x >)$ by a linear combination of step functions having sum of absolute coefficients at most $2\|w\|_1$.

(f)

From (c), we have

$$
S(< w', x > + b') \in cl(\{\tau(< w, x > + b) \text{ for some } w, b\})
$$

and from (d), we have $\forall\ c(y')$ can be approximated by

$$
g(y') = \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} [c(y_i) - c(y_{i-1})] S(y' - x_{i-1})
$$

Thus, $\dfrac{c(y')}{2\|w\|_1}$ can be approximated by

$$
g(y') = \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} \frac{c(y_i) - c(y_{i-1})}{2\|w\|_1} S(y' - x_{i-1})
$$

and

$$
\begin{aligned}
\left| \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} \frac{c(y_i) - c(y_{i-1})}{2\|w\|_1} \right| &\leqslant \frac{1}{2\|w\|_1} \sum_{\substack{i=1 \\ y_{i-1} < y_i}}^{n} |c(y_i) - c(y_{i-1})| \\
&\leqslant 1
\end{aligned}
$$

i.e., it is bounded.

I.e., $\forall\ f \in \mathscr{F}$, f is linear combination of step functions and the sum of coefficients equals 1.

Therefore,

$$
\mathscr{F} \subseteq \overline{conv}\{\tau(< w, x > + b) \text{ for some } w, b\}
$$

(g)

$\because$

$$\begin{aligned}
\mathbb{E}G &= \sum_{i=1}^{m} c_i g_i^* \\
&= f \\
Var(G) &= \mathbb{E}G^2 - (\mathbb{E}G)^2 \\
&\leqslant \mathbb{E}G^2 \\
&= \sum_{i=1}^{m} c_i g_i^{*2} \\
&\leqslant \sum_{i=1}^{m} c_i \\
&= 1
\end{aligned}$$

$\therefore$

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{p}\sum_{i=1}^{p} G_i\right] &= \frac{1}{p}\sum_{i=1}^{p}\mathbb{E}(G_i) \\
&= f \\
Var\left[\frac{1}{p}\sum_{i=1}^{p} G_i\right] &= \frac{1}{p^2}\sum_{i=1}^{p} VarG_i \\
&\leqslant \frac{1}{p}
\end{aligned}$$

$\therefore$

$$\begin{aligned}
\mathbb{E}[\|f_p - f\|^2] &= \mathbb{E}\left\{\int_{x\in[0,1]^d}\left[\frac{1}{p}\sum_{i=1}^{p} G_i - f(x)\right]^2 \mathrm{d}x\right\} \\
&= \int_{x\in[0,1]^d}\mathbb{E}\left[\frac{1}{p}\sum_{i=1}^{p} G_i - f(x)\right]^2 \mathrm{d}x \\
&= \int_{x\in[0,1]^d} Var\left[\frac{1}{p}\sum_{i=1}^{p} G_i\right]\mathrm{d}x \\
&\leqslant \int_{x\in[0,1]^d}\frac{1}{p}\mathrm{d}x \\
&= \frac{1}{p}
\end{aligned}$$

(h)

$\because$ from (f) we have

$$\mathscr{F}_{\cos} \subseteq \overline{conv}(\{\tau(< w, x > +b) \text{ for some } w, b\})$$

$\therefore$ $\forall\ \epsilon > 0$, $\exists\ c_1, \cdots, c_p \in \mathbb{R}, g_1^*, \cdots, g_p^*$ where $c_i \geqslant 0$, $\sum_{i=1}^{p} c_i = 1$ and $g_i^* = \tau(< w_i, x > +b_i)$ for some w_i, b_i, s.t. $g = \sum_{i=1}^{p} c_i g_i^*$, $|f - g| < \epsilon$

Solution (cont.)

$\because$

$$\begin{aligned}\mathbb{E}[\|f_p - f\|^2] &\leqslant \mathbb{E}\left[\|f_p - g\|^2\right] + \mathbb{E}\left[\|\epsilon\|^2\right] \\ &\leqslant \frac{1}{p} + \epsilon\end{aligned}$$

$\therefore$

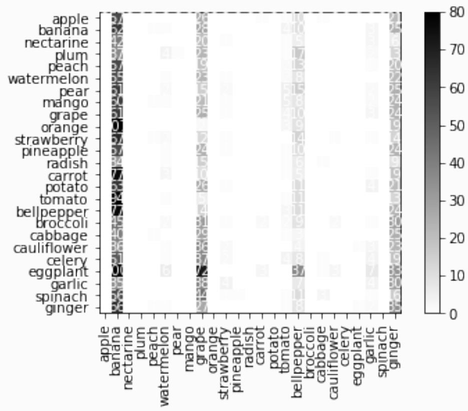
$$\begin{aligned}E(f,p) &= \inf_{h \in \mathscr{H}_p} \int_{x \in [0,1]^d} [f(x) - h(x)]^2 \mathrm{d}x \\ &\leqslant \mathbb{E}[\|f_p - f\|^2] + \epsilon \\ &\leqslant \frac{1}{p} + \epsilon\end{aligned}$$

Therefore,

$$E(f,p) \leqslant \frac{1}{p}$$

Question 3

(a)

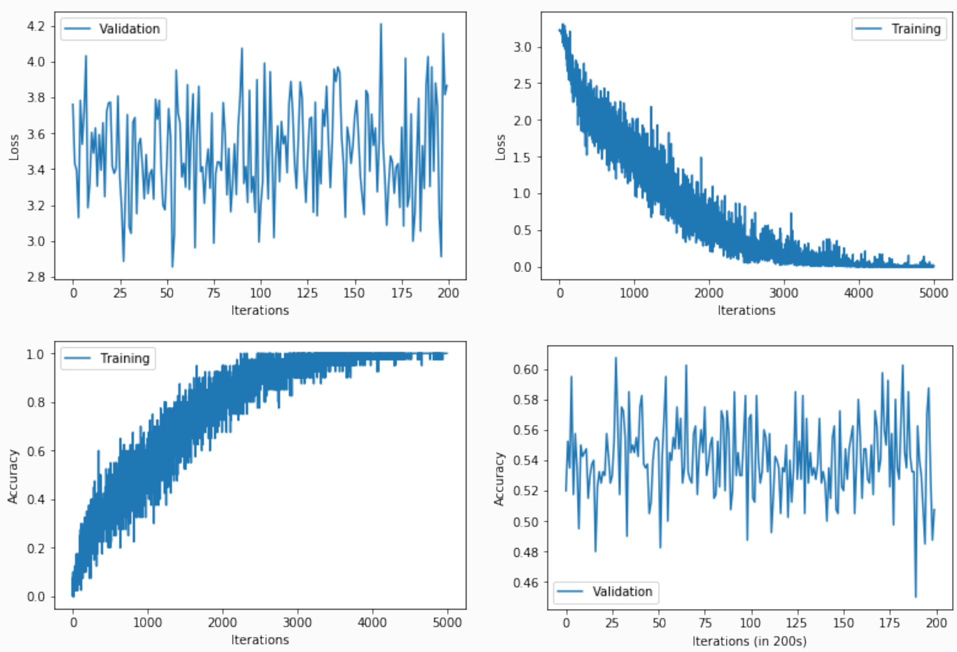



```
def build_network(self,
                  images,
                  num_outputs,
                  scope='yolo'):

with tf.variable_scope(scope):
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        weights_initializer=
                          tf.truncated_normal_initializer(0.0, 0.01),
                        weights_regularizer=
                          slim.l2_regularizer(0.0005)):

        self.conv1 = slim.conv2d(images, 5, [15, 15],
          activation_fn = None, scope='conv1')
        relu1 = tf.nn.relu(self.conv1)
        pool1 = slim.max_pool2d(relu1, [3,3], scope='pool1')
        fc2 = slim.fully_connected(slim.flatten(pool1),
          512, activation_fn = None, scope='fc2')
        relu2 = tf.nn.relu(fc2)
        net = slim.fully_connected(relu2, 25,
          activation_fn = None, scope='fc3')

return net
```

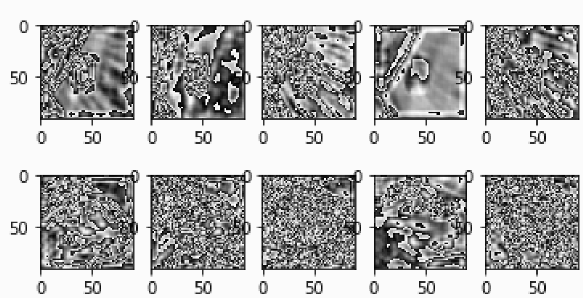
(b)

```python
def get_train_batch(self):
  batch = np.random.choice(self.train_data, self.batch_size)
  features = np.array([i['features'] for i in batch])
  labels = np.array([i['label'] for i in batch])
  return features, labels

def get_validation_batch(self):
    batch = np.random.choice(self.val_data, self.val_batch_size)
    features = np.array([i['features'] for i in batch])
    labels = np.array([i['label'] for i in batch])
    return features, labels
```

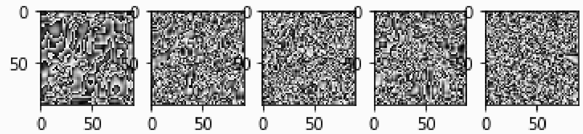
(c)



(d)

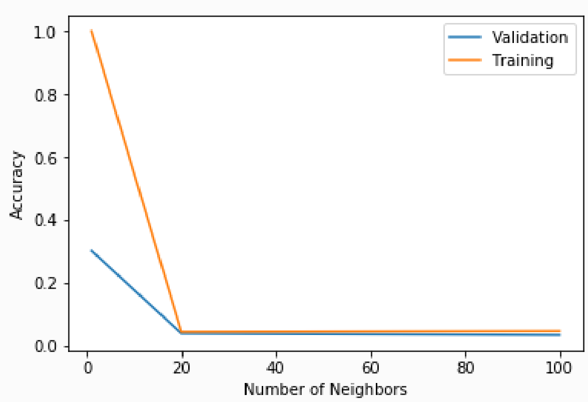


Solution (cont.)



```python
def vizualize_features(self,net):

    images = [0,10,100]
    for j in images:
        batch_eval = np.zeros(
          [1,self.train_data[0]['features'].shape[0],
          self.train_data[0]['features'].shape[1],
          self.train_data[0]['features'].shape[2]])
        batch_eval[0,:,:,:] = self.train_data[j]['features']

        response_map = self.sess.run(net.conv1,
                                     feed_dict={net.images: batch_eval})

        for i in range(5):
            plt.subplot(1,5,i+1)
            img = self.revert_image(response_map[0,:,:,i])
            plt.imshow(img)
        plt.show()
```

(e)




```python
class NN():
    def __init__(self,train_data,val_data,n_neighbors=5):
        self.train_data = train_data
        self.val_data = val_data
```

Solution (cont.)

```python

        self.sample_size = 400
        self.model = KNeighborsClassifier(n_neighbors=n_neighbors)

    def train_model(self):
        x = np.array([np.reshape(self.train_data[i]['features'],
          (90*90*3)) for i in range(len(self.train_data))])
        y = np.array([self.train_data[i]['label'] for i in
          range(len(self.train_data))])
        self.model.fit(x,y)

    def get_validation_error(self):
        index = np.random.choice(len(self.val_data),self.sample_size,
          replace=False)
        predy = self.model.predict([np.reshape(
          self.val_data[i]['features'],
          (90*90*3)) for i in index])
        return np.mean(np.argmax(predy,1)!=np.argmax([
          self.val_data[i]['label'] for i in index],1))

    def get_train_error(self):
        index = np.random.choice(len(self.train_data),
          self.sample_size,replace=False)
        predy = self.model.predict([np.reshape(
          self.train_data[i]['features'],
          (90*90*3)) for i in index])
        return np.mean(np.max(predy,1)!=np.argmax([
          self.train_data[i]['label'] for i in index],1))
```

Question 4

Question What are the regularization methods in CNN?
Solution Empirical

(1) Dropout

Because a fully connected layer occupies most of the parameters, it is prone to overfitting. One method to reduce overfitting is dropout. At each training stage, individual nodes are either "dropped out" of the net with probability $1-p$ or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights.

In the training stages, the probability that a hidden node will be dropped is usually 0.5; for input nodes, this should be much lower, intuitively because information is directly lost when input nodes are ignored.

At testing time after training has finished, we would ideally like to find a sample average of all possible 2^n dropped-out networks; unfortunately this is unfeasible for large values of n. However, we can find an approximation by using the full network with each node's output weighted by a factor of p, so the expected value of the output of any node is the same as in the training stages. This is the biggest contribution of the dropout method: although it effectively generates 2^n neural nets, and as such allows for model combination, at test time only a single network needs to be tested.

By avoiding training all nodes on all training data, dropout decreases overfitting in neural nets. The method also significantly improves the speed of training. This makes model combination practical, even for deep neural nets. The technique seems to reduce node interactions, leading them to learn more robust features that better generalize to new data.

(2) DropConnect

DropConnect is the generalization of dropout in which each connection, rather than each output unit, can be dropped with probability $1-p$. Each unit thus receives input from a random subset of units in the previous layer.

DropConnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output vectors of a layer. In other words, the fully connected layer with DropConnect becomes a sparsely connected layer in which the connections are chosen at random during the training stage.

(3) Stochastic pooling

A major drawback to Dropout is that it does not have the same benefits for convolutional layers, where the neurons are not fully connected.

In stochastic pooling, the conventional deterministic pooling operations are replaced with a stochastic procedure, where the activation within each pooling region is picked randomly according to a multinomial distribution, given by the activities within the pooling region. The approach is hyperparameter free and can be combined with other regularization approaches, such as dropout and data augmentation.

An alternate view of stochastic pooling is that it is equivalent to standard max pooling but with many copies of an input image, each having small local deformations. This is similar to explicit elastic deformations of the input images, which delivers excellent MNIST performance. Using stochastic pooling

Solution (cont.)

in a multilayer model gives an exponential number of deformations since the selections in higher layers are independent of those below.

(4) Artificial data

Since the degree of model overfitting is determined by both its power and the amount of training it receives, providing a convolutional network with more training examples can reduce overfitting. Since these networks are usually trained with all available data, one approach is to either generate new data from scratch (if possible) or perturb existing data to create new ones. For example, input images could be asymmetrically cropped by a few percent to create new examples with the same label as the original.

Explicit

(1) Early stopping

One of the simplest methods to prevent overfitting of a network is to simply stop the training before overfitting has had a chance to occur. It comes with the disadvantage that the learning process is halted.

(2) Number of parameters

Another simple way to prevent overfitting is to limit the number of parameters, typically by limiting the number of hidden units in each layer or limiting network depth. For convolutional networks, the filter size also affects the number of parameters. Limiting the number of parameters restricts the predictive power of the network directly, reducing the complexity of the function that it can perform on the data, and thus limits the amount of overfitting. This is equivalent to a "zero norm".

(3) Weight decay

A simple form of added regularizer is weight decay, which simply adds an additional error, proportional to the sum of weights (L1 norm) or squared magnitude (L2 norm) of the weight vector, to the error at each node. The level of acceptable model complexity can be reduced by increasing the proportionality constant, thus increasing the penalty for large weight vectors.

L2 regularization is the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors. Due to multiplicative interactions between weights and inputs this has the appealing property of encouraging the network to use all of its inputs a little rather than some of its inputs a lot.

L1 regularization is another common form. It is possible to combine L1 with L2 regularization (this is called Elastic net regularization). The L1 regularization leads the weight vectors to become sparse during optimization. In other words, neurons with L1 regularization end up using only a sparse subset of their most important inputs and become nearly invariant to the noisy inputs.

(4) Max norm constraints

Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint. In practice, this corresponds to performing the parameter update as normal, and then enforcing the constraint by clamping the weight vector $\vec{w}$ of every neuron to satisfy $\|\vec{w}\|_2 < c$. Typical values of c are order of 3-4. Some papers report improvements[48] when using this form of regularization.

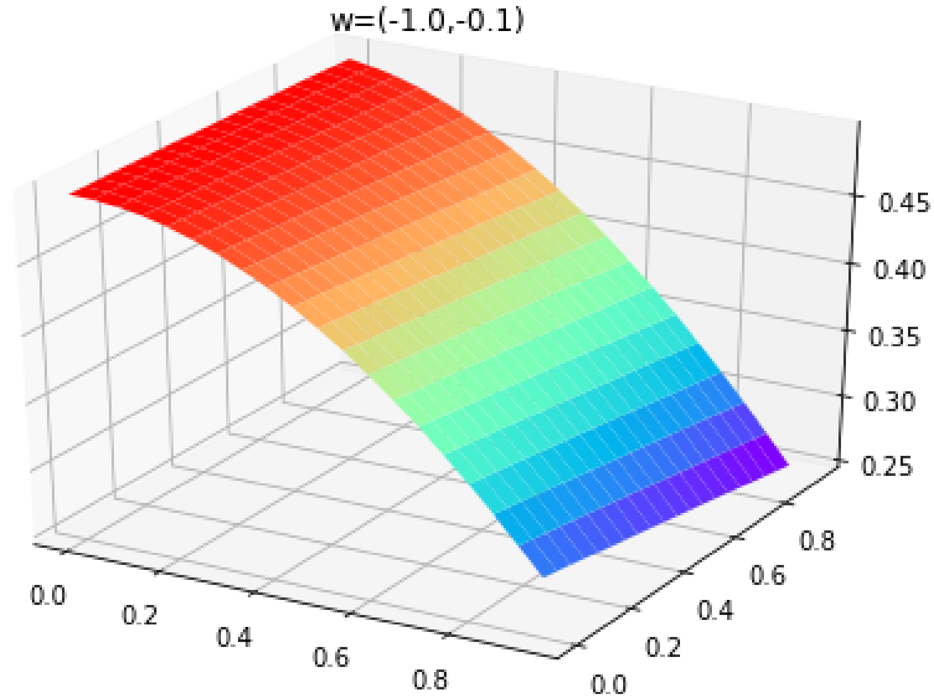
HW13

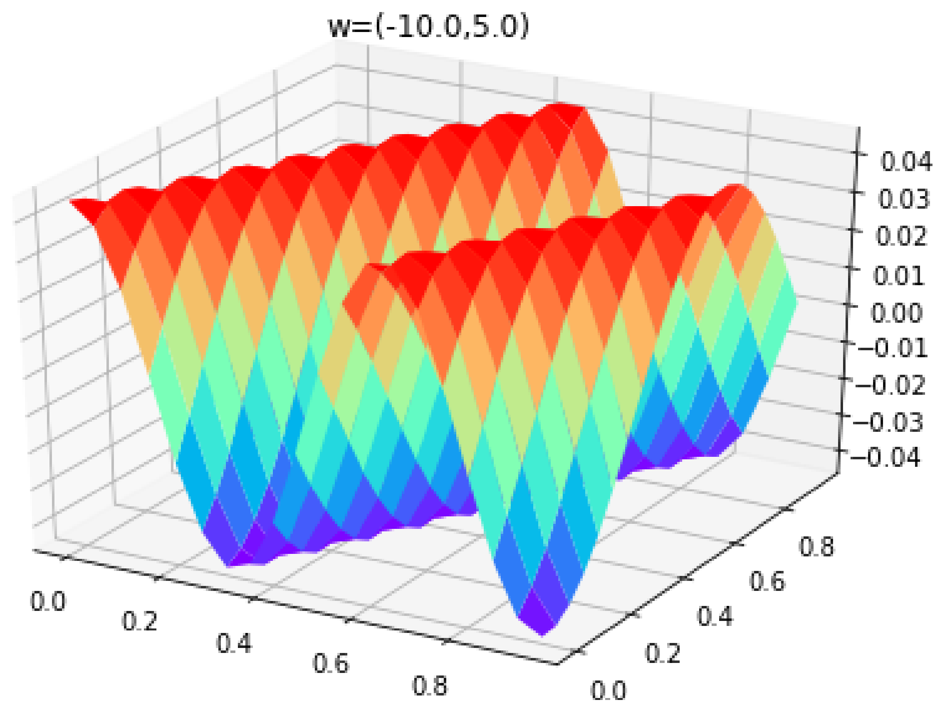
December 1, 2017

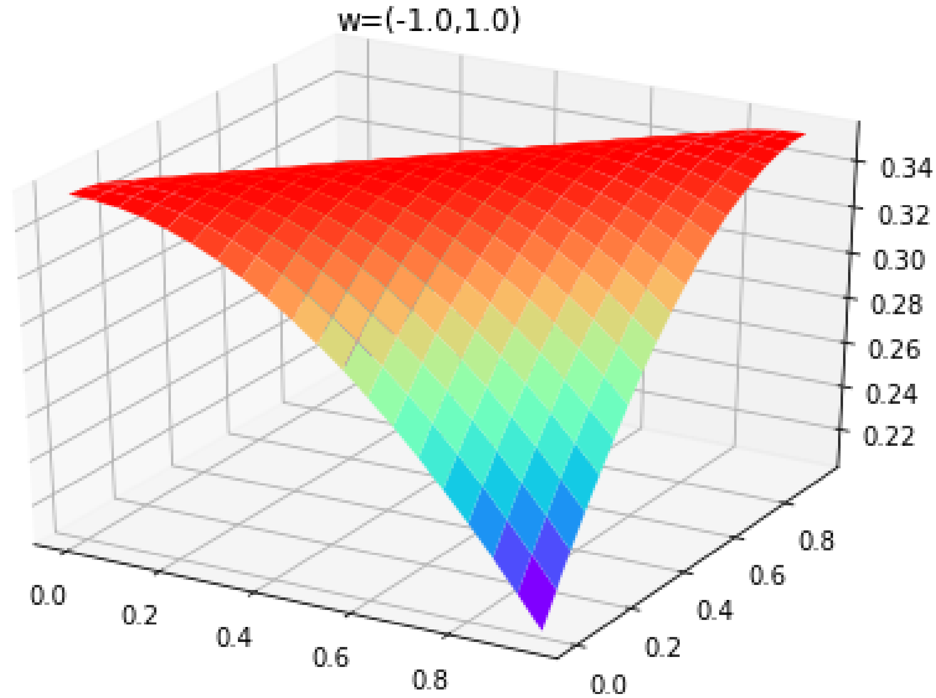
1 Question 2

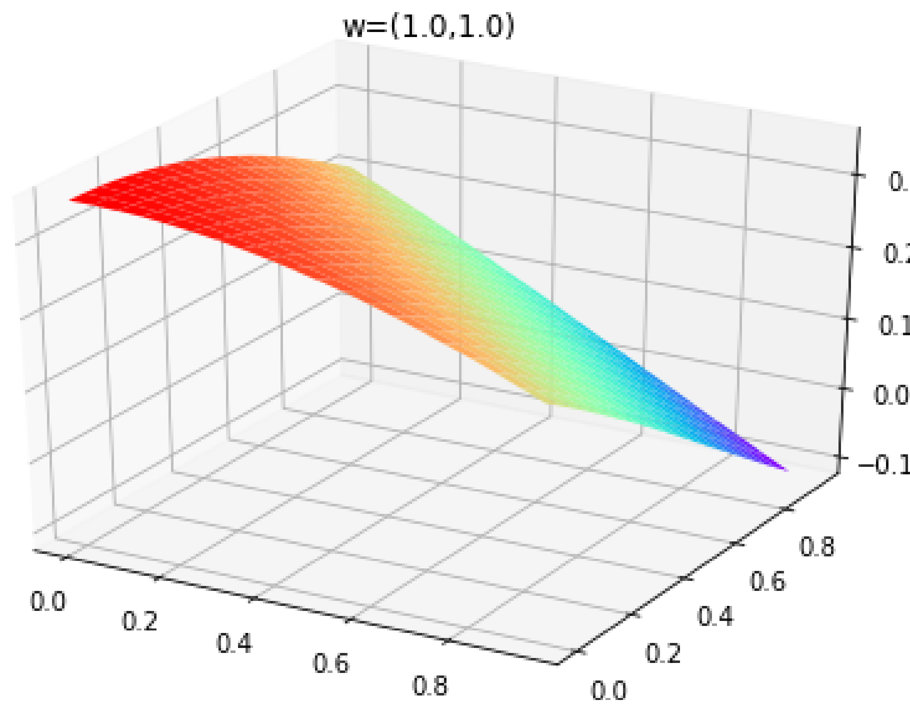
1.0.1 (b)

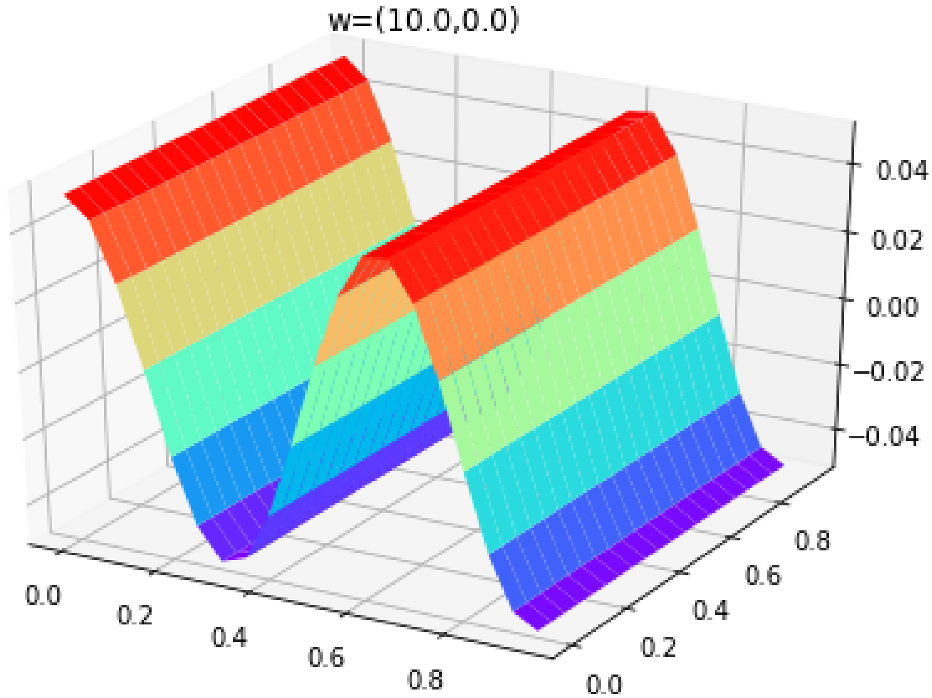
```
In [1]: from matplotlib import pyplot as plt
        import numpy as np
        from mpl_toolkits.mplot3d import Axes3D

In [2]: d = 2
        w = np.array([
            [-1,-0.1],
            [-10,5],
            [-1,1],
            [1,1],
            [10,0]])

In [3]: for i in range(len(w)):
            fig = plt.figure()
            ax = Axes3D(fig)
            X = np.arange(0, 1, 0.05)
            Y = np.arange(0, 1, 0.05)
            X, Y = np.meshgrid(X, Y)
            Z = np.cos(X*w[i,0]+Y*w[i,1])/2/np.linalg.norm(w[i])

            ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='rainbow')
            plt.title(r'w=(%.1f,%.1f)'%(w[i,0],w[i,1]))
            #plt.title(r'$$w=(%f,%f)$$'%(w[i,0],w[i,1]))
            plt.savefig('2b'+str(i))
            plt.show()
```


w=(-1.0,-0.1)
0.45
0.40
0.35
0.30
0.25
0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8


w=(-10.0,5.0)
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8


w=(-1.0,1.0)
0.34
0.32
0.30
0.28
0.26
0.24
0.22
0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8


w=(1.0,1.0)
0.3
0.2
0.1
0.0
−0.1
0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8



2 Question 3

2.1 (a)

data_manager.py

```
In [2]: import os
        import numpy as np
        import copy
        import glob
        import pickle
        import IPython
        from matplotlib import pyplot as plt
        from PIL import Image
        class data_manager(object):
            def __init__(self,classes,image_size,compute_features = None, compute_label = None):

                #Batch Size for training
                self.batch_size = 40
                #Batch size for test, more samples to increase accuracy
                self.val_batch_size = 400

                self.classes = classes
```

```
        self.num_class = len(self.classes)
        self.image_size = image_size

        self.class_to_ind = dict(zip(self.classes, range(len(self.classes))))

        self.cursor = 0
        self.t_cursor = 0
        self.epoch = 1

        self.recent_batch = []

        if compute_features == None:
            self.compute_feature = self.compute_features_baseline

        else:
            self.compute_feature = compute_features

        if compute_label == None:
            self.compute_label = self.compute_label_baseline
        else:
            self.compute_label = compute_label


        self.load_train_set()
        self.load_validation_set()


    def get_train_batch(self):

        '''

        Compute a training batch for the neural network
        The batch size should be size 40

        '''
        batch = np.random.choice(self.train_data,self.batch_size)
        features = np.array([i['features'] for i in batch])
        labels = np.array([i['label'] for i in batch])
        return features,labels


    def get_empty_state(self):
        images = np.zeros((self.batch_size, self.image_size,self.image_size,3))
        return images

    def get_empty_label(self):
        labels = np.zeros((self.batch_size, self.num_class))
        return labels
```

```python
def get_empty_state_val(self):
    images = np.zeros((self.val_batch_size, self.image_size,self.image_size,3))
    return images

def get_empty_label_val(self):
    labels = np.zeros((self.val_batch_size, self.num_class))
    return labels



def get_validation_batch(self):

    '''
    Compute a training batch for the neural network

    The batch size should be size 400

    '''
    #FILL IN
    batch = np.random.choice(self.val_data,self.val_batch_size)
    features = np.array([i['features'] for i in batch])
    labels = np.array([i['label'] for i in batch])
    return features,labels

def compute_features_baseline(self, image):
    '''
    computes the featurized on the images. In this case this corresponds
    to rescaling and standardizing.
    '''

    image = image.resize((self.image_size, self.image_size))
    image = (np.array(image) / 255.0) * 2.0 - 1.0

    return image


def compute_label_baseline(self,label):
    '''
    Compute one-hot labels given the class size
    '''

    one_hot = np.zeros(self.num_class)

    idx = self.classes.index(label)

    one_hot[idx] = 1.0
```

```
        return one_hot


    def load_set(self,set_name):

        '''
        Given a string which is either 'val' or 'train', the function should load all th
        data into an

        '''

        data = []
        data_paths = glob.glob(set_name+'/*.png')

        count = 0


        for datum_path in data_paths:

            label_idx = datum_path.find('_')


            label = datum_path[len(set_name)+1:label_idx]

            if self.classes.count(label) > 0:

                img = Image.open(datum_path)

                label_vec = self.compute_label(label)

                features = self.compute_feature(img)


                data.append({'c_img': np.array(img), 'label': label_vec, 'features': fea

        np.random.shuffle(data)
        return data


    def load_train_set(self):
        '''
        Loads the train set
        '''

        self.train_data = self.load_set('train')


    def load_validation_set(self):
```

```
        '''
        Loads the validation set
        '''

        self.val_data = self.load_set('val')
```

In [3]:
```
import numpy as np
import tensorflow as tf
#import yolo.config_card as cfg

import IPython

slim = tf.contrib.slim


class CNN(object):

    def __init__(self,classes,image_size):
        '''
        Initializes the size of the network
        '''

        self.classes = classes
        self.num_class = len(self.classes)
        self.image_size = image_size

        self.output_size = self.num_class
        self.batch_size = 40

        self.images = tf.placeholder(tf.float32, [None, self.image_size,self.image_size,


        self.logits = self.build_network(self.images, num_outputs=self.output_size)

        self.labels = tf.placeholder(tf.float32, [None, self.num_class])

        self.loss_layer(self.logits, self.labels)
        self.total_loss = tf.losses.get_total_loss()
        tf.summary.scalar('total_loss', self.total_loss)

    def build_network(self,
                      images,
                      num_outputs,
                      scope='yolo'):

        with tf.variable_scope(scope):
            with slim.arg_scope([slim.conv2d, slim.fully_connected],
                                weights_initializer=tf.truncated_normal_initializer(0.0,
```

```
                                weights_regularizer=slim.l2_regularizer(0.0005)):

                    '''
                    Fill in network architecutre here
                    Network should start out with the images function
                    Then it should return net
                    '''
                    self.conv1 = slim.conv2d(images, 5, [15, 15], activation_fn = None, scop
                    relu1 = tf.nn.relu(self.conv1)
                    pool1 = slim.max_pool2d(relu1, [3,3], scope='pool1')
                    fc2 = slim.fully_connected(slim.flatten(pool1), 512, activation_fn = Non
                    relu2 = tf.nn.relu(fc2)
                    net = slim.fully_connected(relu2, 25, activation_fn = None, scope='fc3')

            return net



        def get_acc(self,y_,y_out):

            '''
            Fill in a way to compute accurracy given two tensorflows arrays
            y_ (the true label) and y_out (the predict label)
            '''

            cp = tf.equal(tf.argmax(y_out,1), tf.argmax(y_,1))

            ac = tf.reduce_mean(tf.cast(cp, tf.float32))

            return ac

        def loss_layer(self, predicts, classes, scope='loss_layer'):
            '''
            The loss layer of the network, which is written for you.
            You need to fill in get_accuracy to report the performance
            '''
            with tf.variable_scope(scope):

                self.class_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lab

                self.accuracy = self.get_acc(classes,predicts)
```

confusion_mat.py

```
In [4]: from sklearn.metrics import confusion_matrix
        class Confusion_Matrix(object):
```

```
    def __init__(self,val_data,train_data, class_labels,sess):

        self.val_data = val_data
        self.train_data = train_data
        self.CLASS_LABELS = class_labels
        self.sess = sess


    def test_net(self, net):

        true_labels = []
        predicted_labels = []
        error = []
        for datum in self.val_data:

            batch_eval = np.zeros([1,datum['features'].shape[0],datum['features'].shape[
            batch_eval[0,:,:,:] = datum['features']
            batch_label = np.zeros([1,len(self.CLASS_LABELS)])

            batch_label[0,:] = datum['label']

            prediction = self.sess.run(net.logits,
                                       feed_dict={net.images: batch_eval})

            softmax_error = self.sess.run(net.class_loss,
                                       feed_dict={net.images: batch_eval, net.labels: batch_

            error.append(softmax_error)

            class_pred = np.argmax(prediction)
            class_truth = np.argmax(datum['label'])

            true_labels.append(class_truth)
            predicted_labels.append(class_pred)

        self.getConfusionMatrixPlot(true_labels,predicted_labels,self.CLASS_LABELS)


    def vizualize_features(self,net):

        for datum in self.val_data:

            batch_eval = np.zeros([1,datum['features'].shape[0],datum['features'].shape[
            batch_eval[0,:,:,:] = datum['features']


            batch_label = np.zeros([1,len(self.CLASS_LABELS)])
```

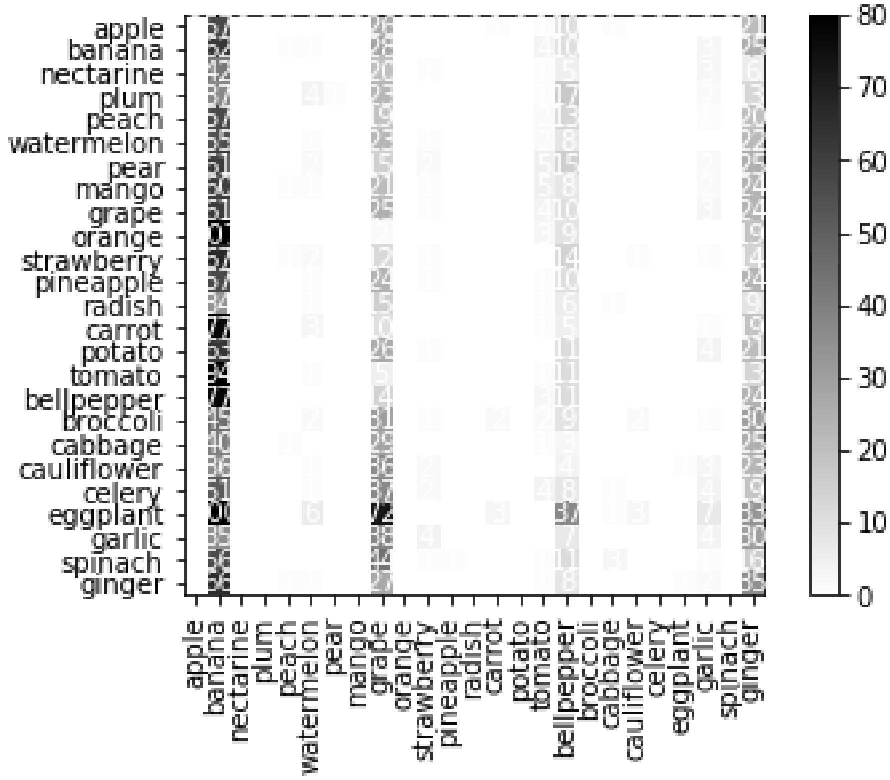
```python
        batch_label[0,:] = datum['label']

        response_map = self.sess.run(net.response_map,
                               feed_dict={net.images: batch_eval, net.labels: batch_

        for i in range(5):
            img = self.revert_image(response_map[0,:,:,i])
            cv2.imshow('debug',img)
            cv2.waitKey(300)



def revert_image(self,img):
    img = (img+1.0)/2.0*255.0

    img = np.array(img,dtype=int)

    blank_img = np.zeros([img.shape[0],img.shape[1],3])

    blank_img[:,:,0] = img
    blank_img[:,:,1] = img
    blank_img[:,:,2] = img

    img = blank_img.astype("uint8")

    return img


def getConfusionMatrix(self,true_labels, predicted_labels):
    """
    Input
    true_labels: actual labels
    predicted_labels: model's predicted labels

    Output
    cm: confusion matrix (true labels vs. predicted labels)
    """

    # Generate confusion matrix using sklearn.metrics
    cm = confusion_matrix(true_labels, predicted_labels)
    return cm


def plotConfusionMatrix(self,cm, alphabet):
    """
    Input
    cm: confusion matrix (true labels vs. predicted labels)
```

```
    alphabet: names of class labels

    Output
    Plot confusion matrix (true labels vs. predicted labels)
    """

    fig = plt.figure()
    plt.clf()                           # Clear plot
    ax = fig.add_subplot(111)           # Add 1x1 grid, first subplot
    ax.set_aspect(1)
    res = ax.imshow(cm, cmap=plt.cm.binary,
                    interpolation='nearest', vmin=0, vmax=80)

    plt.colorbar(res)                   # Add color bar

    width = len(cm)                     # Width of confusion matrix
    height = len(cm[0])                 # Height of confusion matrix

    # Annotate confusion entry with numeric value

    for x in range(width):
        for y in range(height):

            ax.annotate(str(cm[x][y]), xy=(y, x), horizontalalignment='center',
                        verticalalignment='center', color=self.getFontColor(cm[x][y]


    # Plot confusion matrix (true labels vs. predicted labels)
    plt.xticks(range(width), alphabet[:width], rotation=90)
    plt.yticks(range(height), alphabet[:height])
    plt.show()
    return plt


def getConfusionMatrixPlot(self,true_labels, predicted_labels, alphabet):
    """
    Input
    true_labels: actual labels
    predicted_labels: model's predicted labels
    alphabet: names of class labels

    Output
    Plot confusion matrix (true labels vs. predicted labels)
    """

    # Generate confusion matrix using sklearn.metrics
    cm = confusion_matrix(true_labels, predicted_labels)
```

```
        # Plot confusion matrix (true labels vs. predicted labels)
        return self.plotConfusionMatrix(cm, alphabet)


    def getFontColor(self,value):
        """
        Input
        value: confusion entry value

        Output
        font color for confusion entry
        """
        if value < -1:
            return "black"
        else:
            return "white"
```

test_cnn_part_a.py

```
In [26]: tf.reset_default_graph()

         np.random.seed(0)

         CLASS_LABELS = ['apple','banana','nectarine','plum','peach','watermelon','pear','mango'
             'radish','carrot','potato','tomato','bellpepper','broccoli','cabbage','cauliflower'

         image_size = 90
         classes = CLASS_LABELS
         dm = data_manager(classes, image_size)

         cnn = CNN(classes,image_size)

         sess = tf.Session()
         sess.run(tf.global_variables_initializer())

         val_data = dm.val_data
         train_data = dm.train_data

         cm = Confusion_Matrix(val_data,train_data,CLASS_LABELS,sess)
         cm.test_net(cnn)
```



2.2 (c)

trainer.py

```
In [5]: import datetime
        import os
        import sys
        import argparse

        class Solver(object):

            def __init__(self, net, data):


                self.net = net
                self.data = data

                self.max_iter = 5000
                self.summary_iter = 200


                self.learning_rate = 0.1
```

```python
        self.saver = tf.train.Saver()

        self.summary_op = tf.summary.merge_all()

        self.global_step = tf.get_variable(
            'global_step', [], initializer=tf.constant_initializer(0), trainable=False)

        '''
        Tensorflow is told to use a gradient descent optimizer
        In the function optimize you will iteratively apply this on batches of data
        '''
        self.train_step = tf.train.MomentumOptimizer(.003, .9)
        self.train = self.train_step.minimize(self.net.class_loss)


        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())



    def optimize(self):

        self.train_losses = []
        self.test_losses = []
        self.train_accuracy = []
        self.test_accuracy = []
        '''
        Performs the training of the network.
        Implement SGD using the data manager to compute the batches
        Make sure to record the training and test loss through out the process
        '''
        for i in range(self.max_iter):
            x,y = self.data.get_train_batch()
            _,loss,accuracy = self.sess.run([self.train,
                                       self.net.class_loss,
                                       self.net.accuracy],
                                       feed_dict={self.net.images: x, self.net.labels: y})
            self.train_losses.append(loss)
            self.train_accuracy.append(accuracy)

        for i in range(self.summary_iter):
            x,y = self.data.get_validation_batch()
            loss,accuracy = self.sess.run([self.net.class_loss,
                                       self.net.accuracy],
                                       feed_dict={self.net.images: x, self.net.labels: y})
            self.test_losses.append(loss)
            self.test_accuracy.append(accuracy)
```

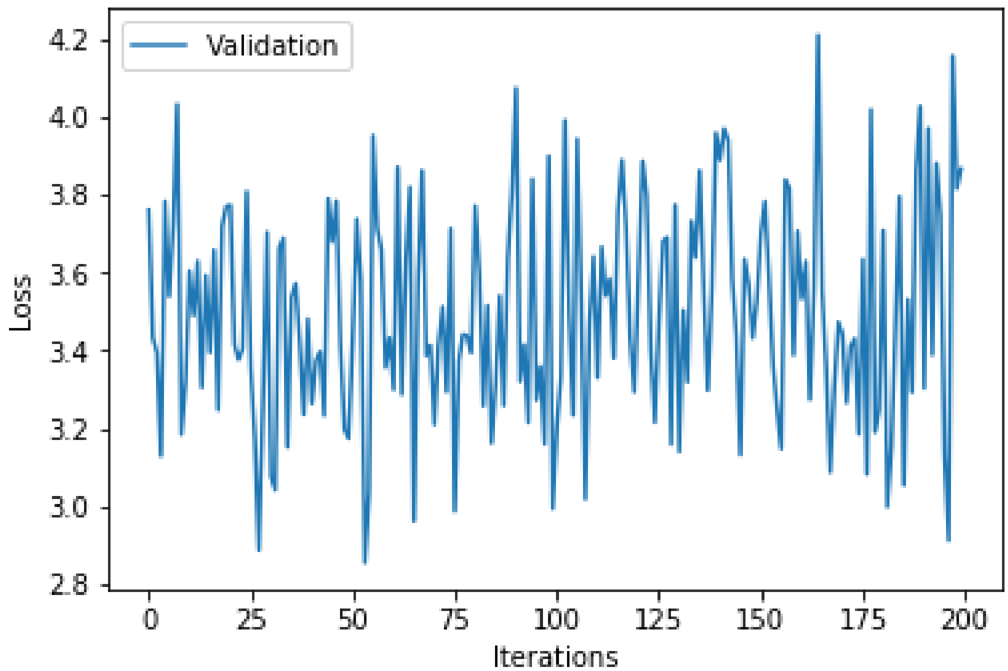
viz_features.py

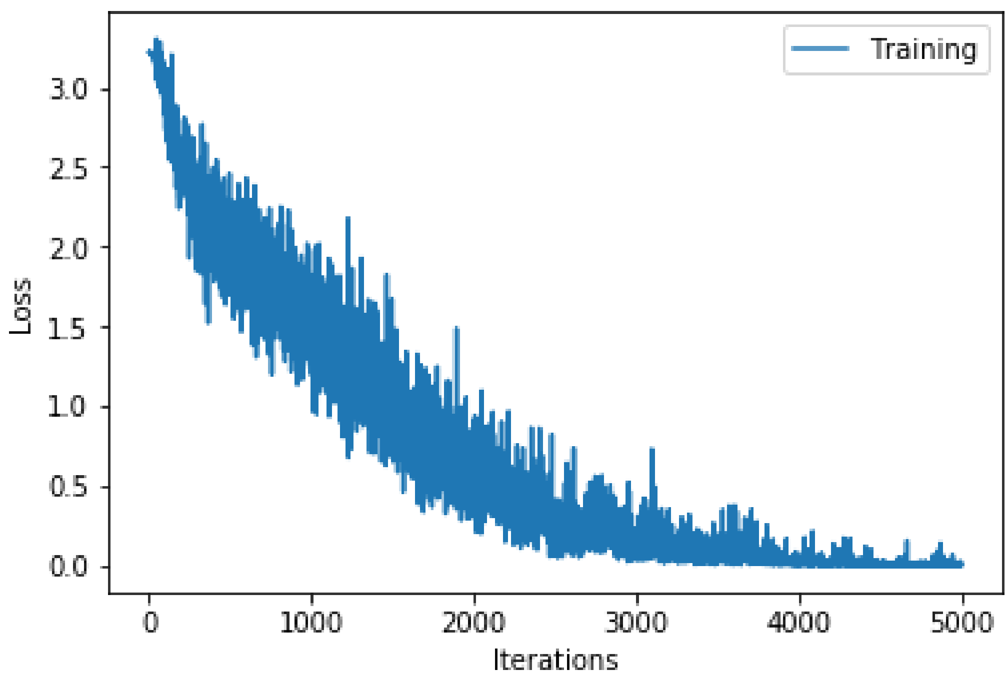
```
In [6]: from sklearn.metrics import confusion_matrix

        class Viz_Feat(object):


            def __init__(self,val_data,train_data, class_labels,sess):

                self.val_data = val_data
                self.train_data = train_data
                self.CLASS_LABELS = class_labels
                self.sess = sess

            def vizualize_features(self,net):

                images = [0,10,100]
                '''
                Compute the response map for the index images
                '''
                for j in images:
                    batch_eval = np.zeros([1,self.train_data[0]['features'].shape[0],
                                           self.train_data[0]['features'].shape[1],
                                           self.train_data[0]['features'].shape[2]])
                    batch_eval[0,:,:,:] = self.train_data[j]['features']

                    response_map = self.sess.run(net.conv1,
                                           feed_dict={net.images: batch_eval})

                    for i in range(5):
                        plt.subplot(1,5,i+1)
                        img = self.revert_image(response_map[0,:,:,i])
                        plt.imshow(img)
                    plt.show()

            def revert_image(self,img):
                '''
                Used to revert images back to a form that can be easily visualized
                '''

                img = (img+1.0)/2.0*255.0

                img = np.array(img,dtype=int)

                blank_img = np.zeros([img.shape[0],img.shape[1],3])

                blank_img[:,:,0] = img
                blank_img[:,:,1] = img
```

```
        blank_img[:,:,2] = img

        img = blank_img.astype("uint8")

        return img
```

train_cnn.py

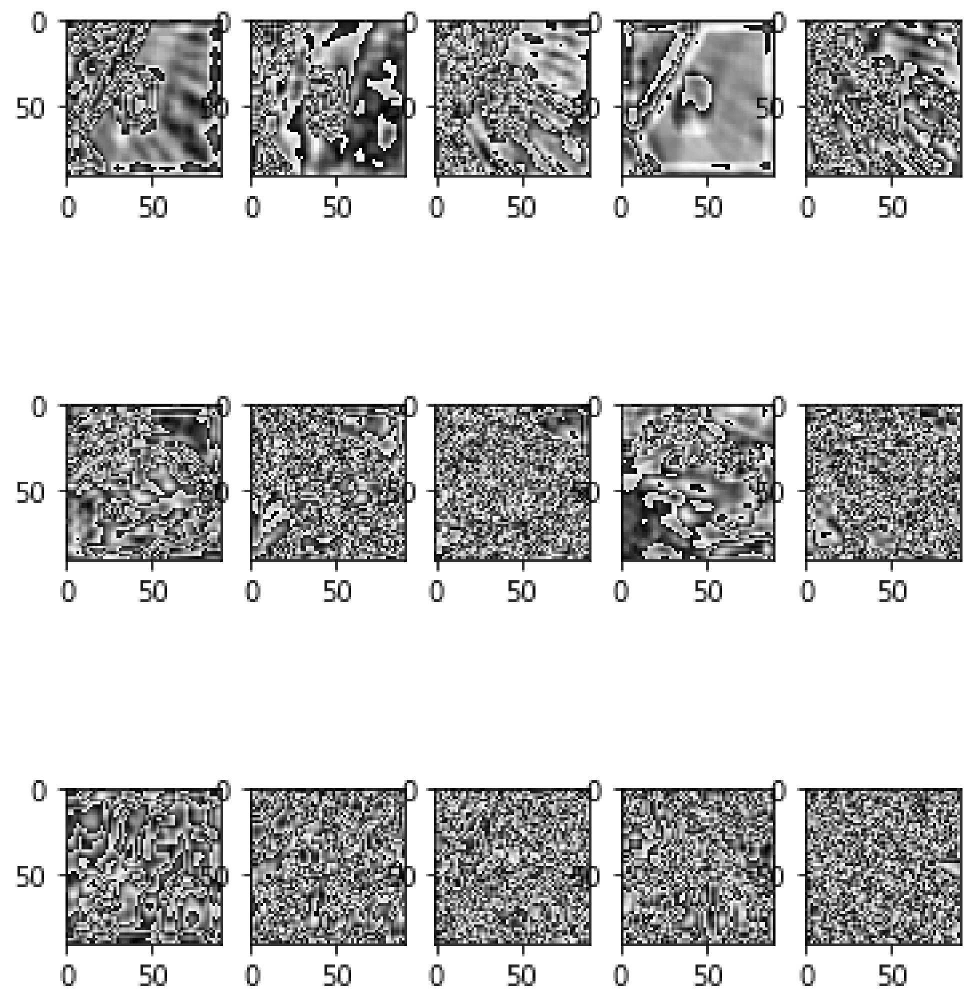
```
In [29]: tf.reset_default_graph()
         CLASS_LABELS = ['apple','banana','nectarine','plum','peach','watermelon','pear','mango'
             'radish','carrot','potato','tomato','bellpepper','broccoli','cabbage','cauliflower'

         LITTLE_CLASS_LABELS = ['apple','banana','eggplant']

         image_size = 90

         np.random.seed(0)

         classes = CLASS_LABELS
         dm = data_manager(classes, image_size)
         with tf.variable_scope("c"):
             cnn = CNN(classes,image_size)

             solver = Solver(cnn,dm)

             solver.optimize()

         plt.plot(solver.test_losses,label = 'Validation')
         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Loss')
         plt.show()
         plt.plot(solver.train_losses, label = 'Training')
         plt.legend()
         plt.xlabel('Iterations')
         plt.ylabel('Loss')
         plt.show()

         val_data = dm.val_data
         train_data = dm.train_data

         sess = solver.sess
```



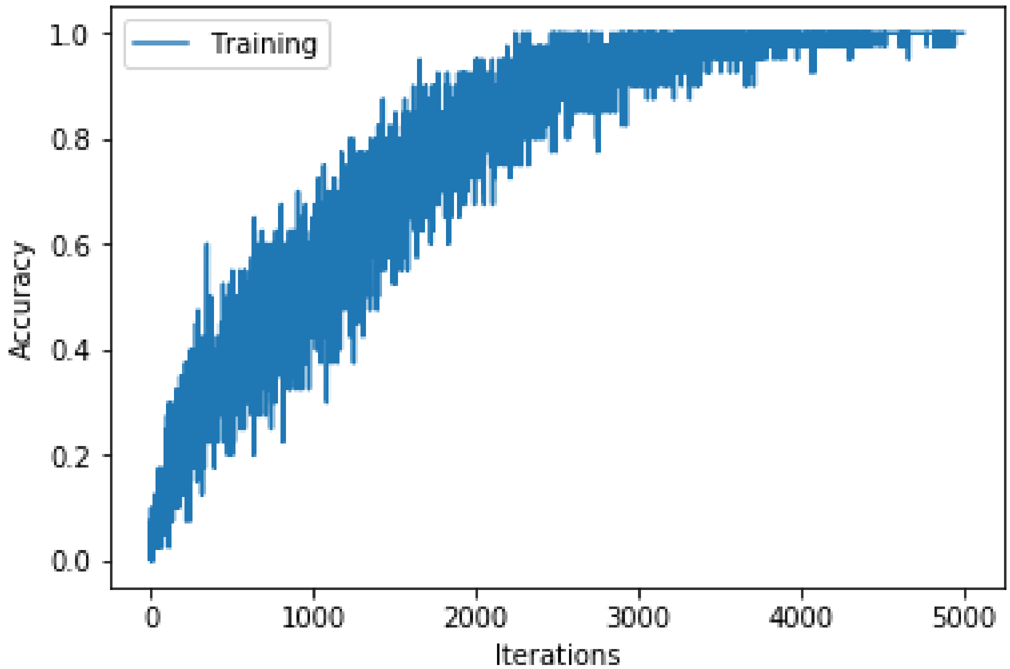



```
In [30]: cm = Viz_Feat(val_data,train_data,CLASS_LABELS,sess)
```

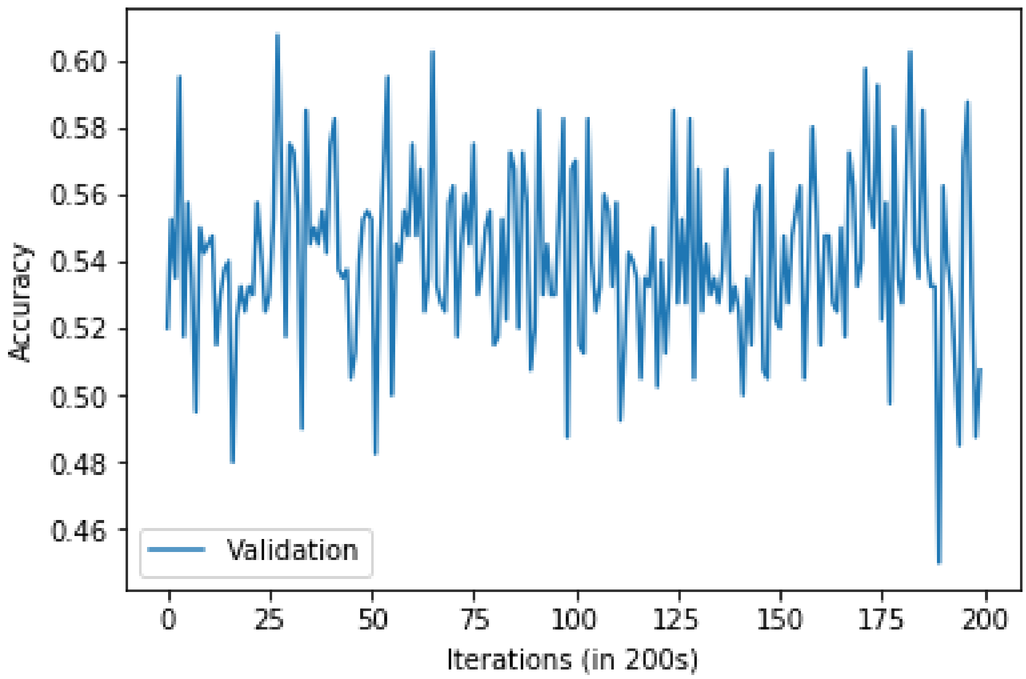
```
cm.vizualize_features(cnn)
```



In [31]:
```
plt.plot(solver.train_accuracy,label = 'Training')
plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.show()
```




```
In [32]: plt.plot(solver.test_accuracy,label = 'Validation')
         plt.legend()
         plt.xlabel('Iterations (in 200s)')
         plt.ylabel('Accuracy')
         plt.show()
```



nn_classifier.py

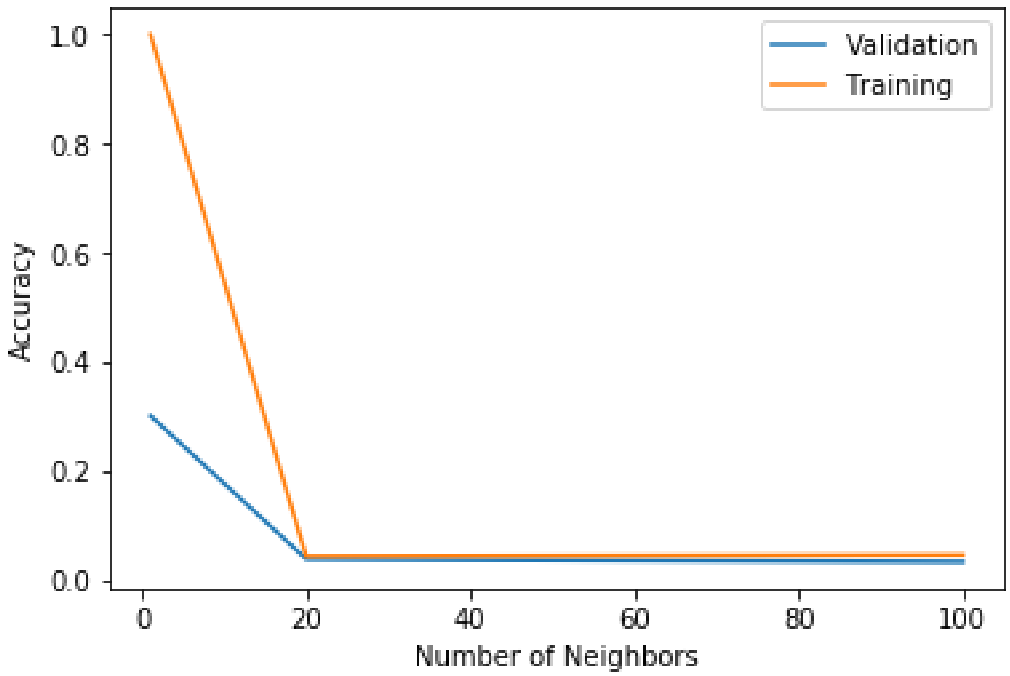
```
In [7]: from numpy.random import uniform
        import time
        import glob
        from  sklearn.neighbors import KNeighborsClassifier

        class NN():
            def __init__(self,train_data,val_data,n_neighbors=5):

                self.train_data = train_data
                self.val_data = val_data

                self.sample_size = 400
                self.model = KNeighborsClassifier(n_neighbors=n_neighbors)


            def train_model(self):

                '''
                Train Nearest Neighbors model
                '''
                x = np.array([np.reshape(self.train_data[i]['features'],(90*90*3)) for i in rang
                y = np.array([self.train_data[i]['label'] for i in range(len(self.train_data))])
                self.model.fit(x,y)
```

```python
    def get_validation_error(self):

        '''
        Compute validation error. Please only compute the error on the sample_size numbe
        over randomly selected data points. To save computation.
        '''
        index = np.random.choice(len(self.val_data),self.sample_size, replace=False)
        predy = self.model.predict([np.reshape(self.val_data[i]['features'],(90*90*3)) f
        return np.mean(np.argmax(predy,1) != np.argmax([self.val_data[i]['label'] for i


    def get_train_error(self):
        '''
        Compute train error. Please only compute the error on the sample_size number
        over randomly selected data points. To save computation.
        '''
        index = np.random.choice(len(self.train_data),self.sample_size, replace=False)
        predy = self.model.predict([np.reshape(self.train_data[i]['features'],(90*90*3))
        return  np.mean(np.argmax(predy,1) != np.argmax([self.train_data[i]['label'] for
```

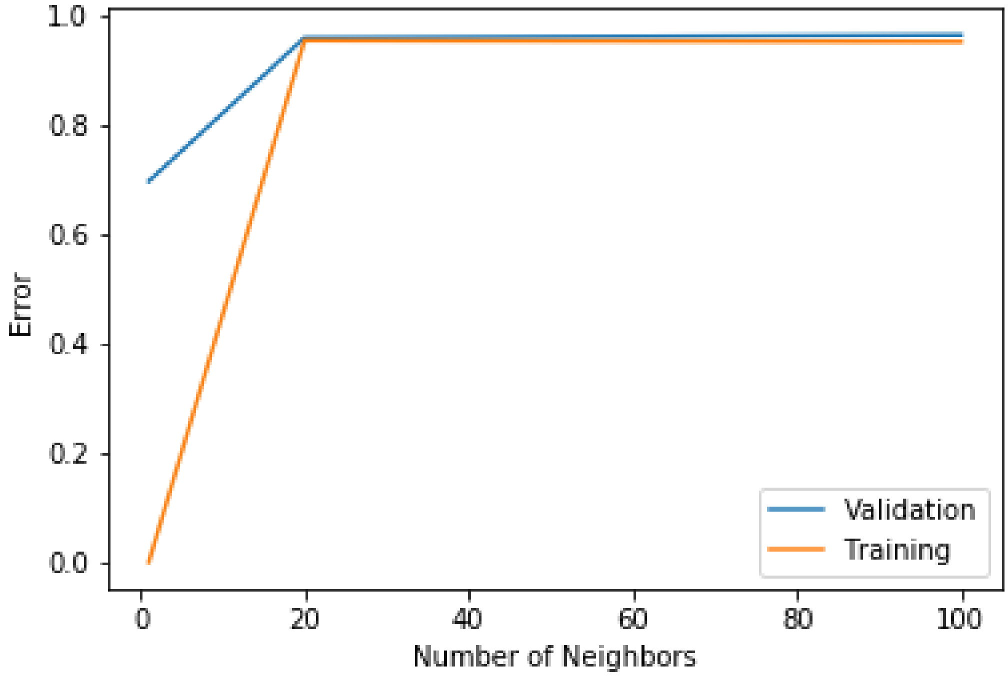
train_nn.py

```python
In [ ]: CLASS_LABELS = ['apple','banana','nectarine','plum','peach','watermelon','pear','mango',
            'radish','carrot','potato','tomato','bellpepper','broccoli','cabbage','cauliflower',


        image_size = 90
        classes = CLASS_LABELS

        dm = data_manager(classes, image_size)

        val_data = dm.val_data
        train_data = dm.train_data

        K = [1, 20, 100]
        test_losses = []
        train_losses = []

        for k in K:
            nn = NN(val_data,train_data,n_neighbors=k)

            nn.train_model()

            test_losses.append(nn.get_validation_error())
            train_losses.append(nn.get_train_error())

In [9]: plt.plot(K, 1-np.array(test_losses),label = 'Validation')
```

```
plt.plot(K, 1-np.array(train_losses), label = 'Training')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```




```
In [11]: plt.plot(K, test_losses,label = 'Validation')
         plt.plot(K, train_losses, label = 'Training')
         plt.legend()
         plt.xlabel('Number of Neighbors')
         plt.ylabel('Error')
         plt.show()
```



1.0
0.8
0.6
0.4
0.2
0.0
Error
0
20
40
60
80
100
Number of Neighbors
Validation
Training


```
In [ ]:
```