The link of my homework is this:
https://github.com/539961733/539961733/tree/master/ECEN%20765/homework3

There are two parts of my code. First part is CNN and another part is SVM.

1. First part

Four python files are needed.

train.py

```
import numpy as np
import tensorflow as tf
from time import time
import math


from include.data import get_data_set
from include.model import model, lr


train_x, train_y = get_data_set("train")
test_x, test_y = get_data_set("test")
tf.set_random_seed(21)
x, y, output, y_pred_cls, global_step, learning_rate = model()
global_accuracy = 0
epoch_start = 0


# PARAMS
_BATCH_SIZE = 128
_EPOCH = 60
_SAVE_PATH = "./tensorboard/cifar-10-v1.0.0/"


# LOSS AND OPTIMIZER
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=output, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9,
                                   beta2=0.999,
                                   epsilon=1e-08).minimize(loss, global_step=global_step)


# PREDICTION AND ACCURACY CALCULATION
correct_prediction = tf.equal(y_pred_cls, tf.argmax(y, axis=1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```python
# SAVER
merged = tf.summary.merge_all()
saver = tf.train.Saver()
sess = tf.Session()
train_writer = tf.summary.FileWriter(_SAVE_PATH, sess.graph)


try:
    print("\nTrying to restore last checkpoint ...")
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=_SAVE_PATH)
    saver.restore(sess, save_path=last_chk_path)
    print("Restored checkpoint from:", last_chk_path)
except ValueError:
    print("\nFailed to restore checkpoint. Initializing variables instead.")
    sess.run(tf.global_variables_initializer())


def train(epoch):
    global epoch_start
    epoch_start = time()
    batch_size = int(math.ceil(len(train_x) / _BATCH_SIZE))
    i_global = 0

    for s in range(batch_size):
        batch_xs = train_x[s*_BATCH_SIZE: (s+1)*_BATCH_SIZE]
        batch_ys = train_y[s*_BATCH_SIZE: (s+1)*_BATCH_SIZE]

        start_time = time()
        i_global, _, batch_loss, batch_acc = sess.run(
            [global_step, optimizer, loss, accuracy],
            feed_dict={x: batch_xs, y: batch_ys, learning_rate: lr(epoch)})
        duration = time() - start_time

        if s % 10 == 0:
            percentage = int(round((s/batch_size)*100))

            bar_len = 29
            filled_len = int((bar_len*int(percentage))/100)
            bar = '=' * filled_len + '>' + '-' * (bar_len - filled_len)

            msg = "Global step: {:>5} - [{}] {:>3}% - acc: {:.4f} - loss: {:.4f} - {:.1f} sample/sec"
            print(msg.format(i_global, bar, percentage, batch_acc, batch_loss, _BATCH_SIZE / duration))
```

```python
        test_and_save(i_global, epoch)


def test_and_save(_global_step, epoch):
    global global_accuracy
    global epoch_start

    i = 0
    predicted_class = np.zeros(shape=len(test_x), dtype=np.int)
    while i < len(test_x):
        j = min(i + _BATCH_SIZE, len(test_x))
        batch_xs = test_x[i:j, :]
        batch_ys = test_y[i:j, :]
        predicted_class[i:j] = sess.run(
            y_pred_cls,
            feed_dict={x: batch_xs, y: batch_ys, learning_rate: lr(epoch)}
        )
        i = j

    correct = (np.argmax(test_y, axis=1) == predicted_class)
    acc = correct.mean()*100
    correct_numbers = correct.sum()

    hours, rem = divmod(time() - epoch_start, 3600)
    minutes, seconds = divmod(rem, 60)
    mes = "\nEpoch {} - accuracy: {:.2f}% ({}/{}) - time: {:0>2}:{:0>2}:{:05.2f}"
    print(mes.format((epoch+1), acc, correct_numbers, len(test_x), int(hours), int(minutes), seconds))

    if global_accuracy != 0 and global_accuracy < acc:

        summary = tf.Summary(value=[
            tf.Summary.Value(tag="Accuracy/test", simple_value=acc),
        ])
        train_writer.add_summary(summary, _global_step)

        saver.save(sess, save_path=_SAVE_PATH, global_step=_global_step)

        mes = "This epoch receive better accuracy: {:.2f} > {:.2f}. Saving session..."
        print(mes.format(acc, global_accuracy))
        global_accuracy = acc

    elif global_accuracy == 0:
        global_accuracy = acc
```

```
print("#####################################################################################
#####################")


def main():
    train_start = time()

    for i in range(_EPOCH):
        print("\nEpoch: {}/{}\n".format((i+1), _EPOCH))
        train(i)

    hours, rem = divmod(time() - train_start, 3600)
    minutes, seconds = divmod(rem, 60)
    mes = "Best accuracy pre session: {:.2f}, time: {:0>2}:{:0>2}:{:05.2f}"
    print(mes.format(global_accuracy, int(hours), int(minutes), seconds))


if __name__ == "__main__":
    main()


sess.close()
```

Predict.py

```
import numpy as np
import tensorflow as tf

from include.data import get_data_set
from include.model import model

test_x, test_y = get_data_set("test")
x, y, output, y_pred_cls, global_step, learning_rate = model()


_BATCH_SIZE = 128
_CLASS_SIZE = 10
_SAVE_PATH = "./tensorboard/cifar-10-v1.0.0/"


saver = tf.train.Saver()
sess = tf.Session()


try:
    print("\nTrying to restore last checkpoint ...")
    last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=_SAVE_PATH)
    saver.restore(sess, save_path=last_chk_path)
    print("Restored checkpoint from:", last_chk_path)
except ValueError:
    print("\nFailed to restore checkpoint. Initializing variables instead.")
    sess.run(tf.global_variables_initializer())


def main():
    i = 0
    predicted_class = np.zeros(shape=len(test_x), dtype=np.int)

    while i < len(test_x):
        j = min(i + _BATCH_SIZE, len(test_x))
        batch_xs = test_x[i:j, :]
        batch_ys = test_y[i:j, :]
        predicted_class[i:j] = sess.run(y_pred_cls, feed_dict={x: batch_xs, y: batch_ys})
        i = j
    correct = (np.argmax(test_y, axis=1) == predicted_class)
    acc = correct.mean() * 100
    correct_numbers = correct.sum()
```

```
    print()
    print("Accuracy on Test-Set: {0:.2f}% ({1} / {2})".format(acc, correct_numbers, len(test_x)))


if __name__ == "__main__":
    main()


sess.close()
```

model.py

```python
import tensorflow as tf


def model():
    _IMAGE_SIZE = 32
    _IMAGE_CHANNELS = 3
    _NUM_CLASSES = 10

    with tf.name_scope('main_params'):
        x = tf.placeholder(tf.float32, shape=[None, _IMAGE_SIZE * _IMAGE_SIZE *
_IMAGE_CHANNELS], name='Input')
        y = tf.placeholder(tf.float32, shape=[None, _NUM_CLASSES], name='Output')
        x_image = tf.reshape(x, [-1, _IMAGE_SIZE, _IMAGE_SIZE, _IMAGE_CHANNELS], name='images')

        global_step = tf.Variable(initial_value=0, trainable=False, name='global_step')
        learning_rate = tf.placeholder(tf.float32, shape=[], name='learning_rate')

    with tf.variable_scope('conv1') as scope:
        conv = tf.layers.conv2d(
            inputs=x_image,
            filters=32,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )
        conv = tf.layers.conv2d(
            inputs=conv,
            filters=64,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
        )
        pool = tf.layers.max_pooling2d(conv, pool_size=[2, 2], strides=2, padding='SAME')
        drop = tf.layers.dropout(pool, rate=0.25, name=scope.name)

    with tf.variable_scope('conv2') as scope:
        conv = tf.layers.conv2d(
            inputs=drop,
            filters=128,
            kernel_size=[3, 3],
            padding='SAME',
            activation=tf.nn.relu
```

```
        )
        pool = tf.layers.max_pooling2d(conv, pool_size=[2, 2], strides=2, padding='SAME')
        conv = tf.layers.conv2d(
            inputs=pool,
            filters=128,
            kernel_size=[2, 2],
            padding='SAME',
            activation=tf.nn.relu
        )
        pool = tf.layers.max_pooling2d(conv, pool_size=[2, 2], strides=2, padding='SAME')
        drop = tf.layers.dropout(pool, rate=0.25, name=scope.name)

    with tf.variable_scope('fully_connected') as scope:
        flat = tf.reshape(drop, [-1, 4 * 4 * 128])

        fc = tf.layers.dense(inputs=flat, units=1500, activation=tf.nn.relu)
        drop = tf.layers.dropout(fc, rate=0.5)
        softmax = tf.layers.dense(inputs=drop, units=_NUM_CLASSES, activation=tf.nn.softmax,
name=scope.name)
        softmax = tf.Print(softmax,[softmax],"Probility of every element is ")

    y_pred_cls = tf.argmax(softmax, axis=1)


    return x, y, softmax, y_pred_cls, global_step, learning_rate


def lr(epoch):
    learning_rate = 1e-3
    if epoch > 80:
        learning_rate *= 0.5e-3
    elif epoch > 60:
        learning_rate *= 1e-3
    elif epoch > 40:
        learning_rate *= 1e-2
    elif epoch > 20:
        learning_rate *= 1e-1
    return learning_rate
```

data.py

```python
import pickle
import numpy as np
import os
from urllib.request import urlretrieve
import tarfile
import zipfile
import sys


def get_data_set(name="train"):
    x = None
    y = None

    maybe_download_and_extract()

    folder_name = "cifar_10"

    f = open('./data_set/'+folder_name+'/batches.meta', 'rb')
    f.close()

    if name is "train":
        for i in range(5):
            f = open('./data_set/'+folder_name+'/data_batch_' + str(i + 1), 'rb')
            datadict = pickle.load(f, encoding='latin1')
            f.close()

            _X = datadict["data"]
            _Y = datadict['labels']

            _X = np.array(_X, dtype=float) / 255.0
            _X = _X.reshape([-1, 3, 32, 32])
            _X = _X.transpose([0, 2, 3, 1])
            _X = _X.reshape(-1, 32*32*3)

            if x is None:
                x = _X
                y = _Y
            else:
                x = np.concatenate((x, _X), axis=0)
                y = np.concatenate((y, _Y), axis=0)

    elif name is "test":
```

```python
        f = open('./data_set/'+folder_name+'/test_batch', 'rb')
        datadict = pickle.load(f, encoding='latin1')
        f.close()

        x = datadict["data"]
        y = np.array(datadict['labels'])

        x = np.array(x, dtype=float) / 255.0
        x = x.reshape([-1, 3, 32, 32])
        x = x.transpose([0, 2, 3, 1])
        x = x.reshape(-1, 32*32*3)

    return x, dense_to_one_hot(y)


def dense_to_one_hot(labels_dense, num_classes=10):
    num_labels = labels_dense.shape[0]
    index_offset = np.arange(num_labels) * num_classes
    labels_one_hot = np.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1

    return labels_one_hot


def _print_download_progress(count, block_size, total_size):
    pct_complete = float(count * block_size) / total_size
    msg = "\r- Download progress: {0:.1%}".format(pct_complete)
    sys.stdout.write(msg)
    sys.stdout.flush()


def maybe_download_and_extract():
    main_directory = "./data_set/"
    cifar_10_directory = main_directory+"cifar_10/"
    if not os.path.exists(main_directory):
        os.makedirs(main_directory)

        url = "http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
        filename = url.split('/')[-1]
        file_path = os.path.join(main_directory, filename)
        zip_cifar_10 = file_path
        file_path, _ = urlretrieve(url=url, filename=file_path, reporthook=_print_download_progress)

        print()
```

```
        print("Download finished. Extracting files.")
        if file_path.endswith(".zip"):
            zipfile.ZipFile(file=file_path, mode="r").extractall(main_directory)
        elif file_path.endswith((".tar.gz", ".tgz")):
            tarfile.open(name=file_path, mode="r:gz").extractall(main_directory)
        print("Done.")

        os.rename(main_directory+"./cifar-10-batches-py", cifar_10_directory)
        os.remove(zip_cifar_10)
```

2. Second Part

Here is SVM part.

```
# Run some setup code for this notebook.

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

from __future__ import print_function

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (12.0, 10.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

from __future__ import print_function

from six.moves import cPickle as pickle
import numpy as np
import os
from scipy.misc import imread
import platform

def load_pickle(f):
    version = platform.python_version_tuple()
    if version[0] == '2':
        return  pickle.load(f)
```

```python
    elif version[0] == '3':
        return   pickle.load(f, encoding='latin1')
    raise ValueError("invalid python version: {}".format(version))

def load_CIFAR_batch(filename):
    """ load single batch of cifar """
    with open(filename, 'rb') as f:
        datadict = load_pickle(f)
        X = datadict['data']
        Y = datadict['labels']
        X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("float")
        Y = np.array(Y)
        return X, Y

def load_CIFAR10(ROOT):
    """ load all of cifar """
    xs = []
    ys = []
    for b in range(1,6):
        f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
        X, Y = load_CIFAR_batch(f)
        xs.append(X)
        ys.append(Y)
    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    del X, Y
    Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
    return Xtr, Ytr, Xte, Yte


# Load the raw CIFAR-10 data.
cifar10_dir = 'data/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 5
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()

# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
```

```
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)

# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)

# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])
```

```python
print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)

class LinearSVM:
    def __init__(self):
        self.W = None

    def loss(self, X, y, reg):
        """
        Structured SVM loss function, vectorized implementation.

        Inputs and outputs are the same as svm_loss_naive.
        """
        loss = 0.0
        dW = np.zeros(self.W.shape) # initialize the gradient as zero

        num_train = X.shape[0]
        scores = X.dot(self.W)
        correct_class_score = scores[range(num_train), list(y)].reshape(-1,1) # (N,1)
        margin = np.maximum(0, scores - correct_class_score + 1)
        margin[range(num_train), list(y)] = 0
        loss = np.sum(margin) / num_train + 0.5 * reg * np.sum(self.W * self.W)

        num_classes = self.W.shape[1]
        inter_mat = np.zeros((num_train, num_classes))
        inter_mat[margin > 0] = 1
        inter_mat[range(num_train), list(y)] = 0
        inter_mat[range(num_train), list(y)] = -np.sum(inter_mat, axis=1)

        dW = (X.T).dot(inter_mat)
        dW = dW/num_train + reg*self.W

        return loss, dW

    def train(self, X, y, learning_rate=1e-3, reg=1e-5, num_iters=100,
              batch_size=200, verbose=False):
        """
        Train this linear classifier using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there are N
          training samples each of dimension D.
        - y: A numpy array of shape (N,) containing training labels; y[i] = c
          means that X[i] has label 0 <= c < C for C classes.
```

```
        - learning_rate: (float) learning rate for optimization.
        - reg: (float) regularization strength.
        - num_iters: (integer) number of steps to take when optimizing
        - batch_size: (integer) number of training examples to use at each step.
        - verbose: (boolean) If true, print progress during optimization.

        Outputs:
        A list containing the value of the loss function at each training iteration.
        """
        num_train, dim = X.shape
        num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
        if self.W is None:
            # lazily initialize W
            self.W = 0.001 * np.random.randn(dim, num_classes)

        # Run stochastic gradient descent to optimize W
        loss_history = []
        for it in range(num_iters):
            X_batch = None
            y_batch = None
            idx_batch = np.random.choice(num_train, batch_size, replace = True)
            X_batch = X[idx_batch]
            y_batch = y[idx_batch]

            # evaluate loss and gradient
            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            self.W -=    learning_rate * grad

            if verbose and it % 100 == 0:
                print('iteration %d / %d: loss %f' % (it, num_iters, loss))

        return loss_history

    def predict(self, X):
        """
        Use the trained weights of this linear classifier to predict labels for
        data points.

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there are N
          training samples each of dimension D.
```

```
        Returns:
        - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
          array of length N, and each element is an integer giving the predicted
          class.
        """
        y_pred = np.zeros(X.shape[0])
        scores = X.dot(self.W)
        y_pred = np.argmax(scores, axis = 1)
        return y_pred

import time
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()

# Write the LinearSVM.predict function and evaluate the performance on both the
# training and validation set
y_train_pred = svm.predict(X_train)
print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
y_val_pred = svm.predict(X_val)
print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.4 on the validation set.
#learning_rates = [1e-7, 5e-5]
#regularization_strengths = [2.5e3, 5e3]
learning_rates = [1.4e-7, 1.5e-7, 1.6e-7]
regularization_strengths = [8000.0, 9000.0, 10000.0, 11000.0, 18000.0, 19000.0, 20000.0, 21000.0]

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation rate.
```

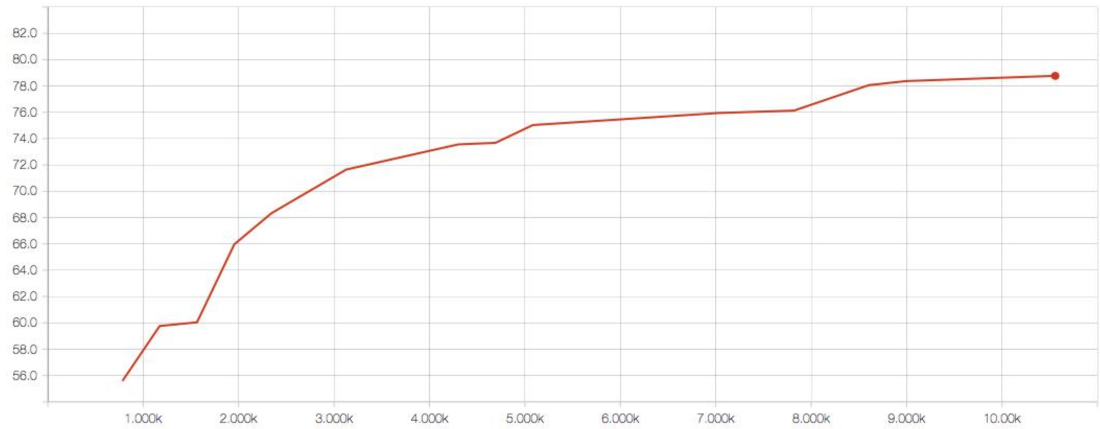
```python
for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        loss = svm.train(X_train, y_train, learning_rate=lr, reg=reg, num_iters=3000)
        y_train_pred = svm.predict(X_train)
        training_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
        results[(lr, reg)] = training_accuracy, val_accuracy
# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)

# Visualize the cross-validation results
import math
x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
```

```python
plt.title('CIFAR-10 validation accuracy')
plt.show()
```

Programming Assignment

1. I chose AlexNet by using TensorFlow in my assignment.
2. For every epoch, I use VPS to run the program and here is the training accuracy.

The final classification accuracy is 78.57% (7857 / 10000).



Because of disadvantage of PyCharm, I can only show the first three probabilities in some test sets.

Probility of every element is [[1.72507109e-23 2.0404182e-27 3.16461514e-28...]...]

Probility of every element is [[5.3257343e-10 2.18905519e-11 5.6569735e-11...]...]

Probility of every element is [[4.30056124e-34 1.07228693e-37 2.42612493e-22...]...]

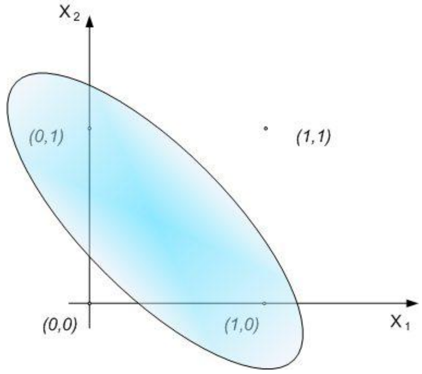
You can see more results by running program. When you run the program, please, running train.py first. This file is used to set up the model by using training data. It can exact file by itself and use the model.py to train the model. You can exit in any epoch because I have set check point. In predict.py, program will use the model which is trained before check point to classify every text data.

3. If you want to see more details and results in SVM program, you can use the link to see the results. Training accuracy is 0.385796 and the validation accuracy is 0.379000. We can find out that SVM is much worse than CNN. However, SVM is machine learning program so it can cost much less time than CNN.

Math Problem

1)

Linear Separity can be no longer used with XOR function.It means that it's not possible to find a line which separates data space to space with output signal - 0, and space with output signal - 1 . Inside the oval area signal on output is '1'. Outside of this area, output signal equals '0'. It's not possible to make it by one line.



The coefficients of this line and the weights W11, W12 and b1make no affect to impossibility of using linear separity. So we can't implement XOR function by one perceptron.

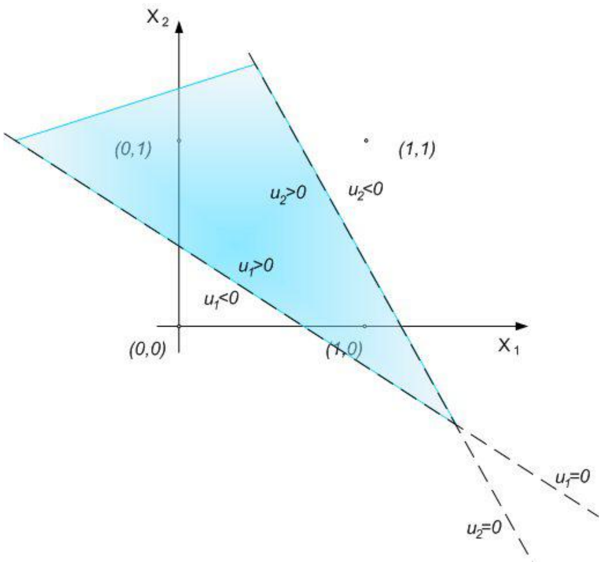
The solve of this problem is an extension of the network in the way that one added neuron in the layer creates new network. Neurons in this network have weights that implement division of space as below:

For 1st neuron

$$u1 = W11x1 + W12x\ 2 + b1 > 0$$
$$u1 = W21x1 + W22x\ 2 + b1 < 0$$

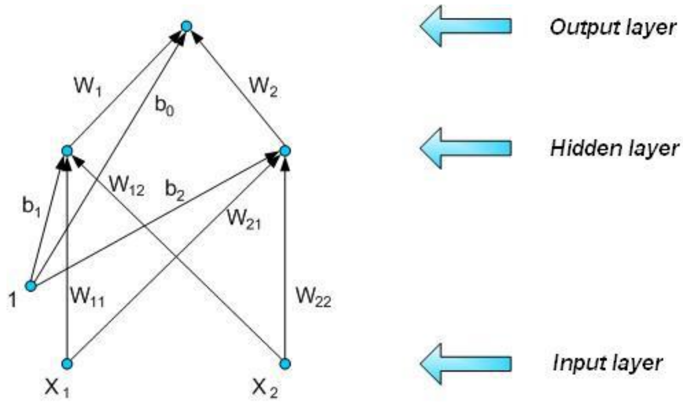
For 2nd neuron

$$u2 = W21x1 + W22x\ 2 + b2 > 0$$
$$u2 = W21x1 + W22x\ 2 + b2 < 0$$



After adding the next layer with neuron, it's possible to make logical sum. On the figure we can see it as a common area of sets u1>0 and u2>0.

Next figure shows full multilayer neural network structure that can implement XOR function. Each additional neuron makes possible to create linear division on ui>0 and ui<0 border that depends on neuron weights. Output layer is the layer that is combination of smaller areas in which was divided input area (by additional neuron).



2)

In more practical terms, VAEs represent latent space (bottleneck layer) as a Guassian random variable (enabled by a constraint on the loss function). Hence, the loss function for the VAEs consist of two terms: a reconstruction loss forcing the decoded samples to match the initial inputs (just like in our previous autoencoders), and the KL divergence between the learned latent distribution and the prior distribution, acting as a regularization term.

$$\min L(x, x') = \min E(x, x') + KL(q(z/x) \,||\, p(z))$$

Here, the first term is the reconstruction loss as before (in a typical auto-encoder). The second term is the Kullback-Leibler divergence between the encoder's distribution, q(z\vert x)q(z∣x) and the true posterior p(z)p(z), typically a Guassian.

As typically (especially for images) the binary cross-entropy is used as the reconstruction loss term, the above loss term for the VAEs can be written as

$$\min L(x, x') = \min - E_{z \sim q(z/x)}[\log p(x'/z)] + KL(q(z/x) \,||\, p(z))$$

To summarize a typical implementation of a VAE, first, an encoder network turns the input samples xx into two parameters in a latent space, **z_mean** and **z_log_sigma**. Then, we randomly sample similar points zz from the latent normal distribution that is assumed to generate the data, via **Z= z_mean + exp(z_log_sigma)** * ϵ, where ϵ is a random normal tensor. Finally, a decoder network maps these latent space points back to the original input data.