Machine Learning II
Prof. Jafari
Jue Wang, Tianyi Wang, Yina Bao
December 4th, 2018

Google Street View Images Project Final Report

Introduction

In this project, we used street view house numbers dataset to explore how computer recognizes digits numbers from images. For this real-world image dataset, we engage in can be applied in image recognition and classification from image to digits which is an important topic in machine learning field. This real-world street view house number image dataset is different than the other dataset because it has over 600,000-digit images and the numbers are in the natural scene images. SVHN is obtained from house numbers in Google Street View images. By giving pixel value vectors as features, we are leveraging ten classification models to predict the label of images. In this project, we aimed at applying various machine learning including deep learning techniques in order to do the image classification. The models respectively are Multilayer Perceptron Networks (MLP), Convolutional Neural Network (CNN) on PyTorch and 1 Convolutional Neural Network on Caffe. Though these different models, we can also find out how these algorithms compare to each other and how can we improve our model.

Description of the dataset

Here is the link of the dataset: http://ufldl.stanford.edu/housenumbers/

- The dataset has ten classes, one for each digit. Digit '0' has label 0, '9' has label 9.
- In the dataset, they separate data into two groups: 73,257 digits for training and 26,032 digits for testing. There also have 531,131 additional data to use as extra training data.

Figure. 1 Data images

This data has two formats: one is the original images with character level bounding boxes, and the other one is MNIST-like 32-by-32 color images centered around a single character. In this project, we are going to use the 32-by-32 color images that all images are resized to the fixed resolutions. (*Figure.1*)

Background Information on the Development of the Algorithm

There have some feature representations for SVHN dataset. Previous approaches to class labels or digits from digital images are using hand-crafted features such as Histograms-of-Oriented-Gradients (HOG) and an off-the-shelf cocktail of the binary image (stacked sparse auto-encoders and the K-means-based system of) (Netzer, 4). These two are popular options when one tries to solve digit classification problem by using standard machine learning procedures. However, the problem is in order to use either one of them, one must binarize the input images grayscale. Recent scholar introduces a new approach that using ConvNets. The ConvNet architecture has numbers of convolution module, and it followed by pooling module and a normalization module (Sermanet, 1).

In our project, our team plans to implement the image classification using the aforementioned algorithms. We are going to compare the MLP model with different layers convolutional neural network models on the training dataset and extra training dataset, and then we are going to apply Caffe on the convolutional neural network model 2. Regarding the

accuracy and efficiency based on the performance of the models to be evaluated, and we are optimizing different models in each scenario by tuning the algorithms.

Experimental Setup

As we explained in the data description section, the dataset consists of three parts: training dataset, test dataset, and the extra training dataset. The main part of our project, we choose to use 73,257 images in training dataset to train our model. In addition, we also applied the full extra training dataset, which has 531,131 images, on CNN model 1 and CNN model 4 as references to compare how the accuracy will be different than our main models. In our project, there are lots of techniques for refinement and many hyper-parameters need to be tuned, and then observe the loss and accuracy on the training dataset. After checking the results, we can decide which approach may improve the accuracy on the test dataset. For the CNN models, we mainly adjust in three different ways: add layers, increase or decrease the number of hidden units, change the probability of dropout and learning rate. We also apply the CNN model 2 on Caffe.

Multilayer Perceptron Networks*:*

MLP network has two linear layers and one ReLU layer. The first linear layer has input size as 3*32*32 and output size as hidden_size (=500). The second linear layer has input size as hidden_size and output size as the number of classes (=10).

Convolutional Neural Network (CNN):

- ***Model 1****:* This is the base convolution network which has two feature stages and one fully connected hidden layer. Each convolution stage contains a convolution module, followed by the max-pooling module. The convolution kernel has the size of 5 * 5. The first convolution layer produces 16 feature convolution filters, while the second convolution layer outputs 32 feature filters. Both of convolution layers have zero-padding and zero-stride on the input. Both of convolution layer apply the same max pooling window with the size as 2 * 2 with stride 2 * 2. The fully connected layer has input-feature = 800 (32 * 5 * 5) and output-feature = 10. We define our batch size as 100.

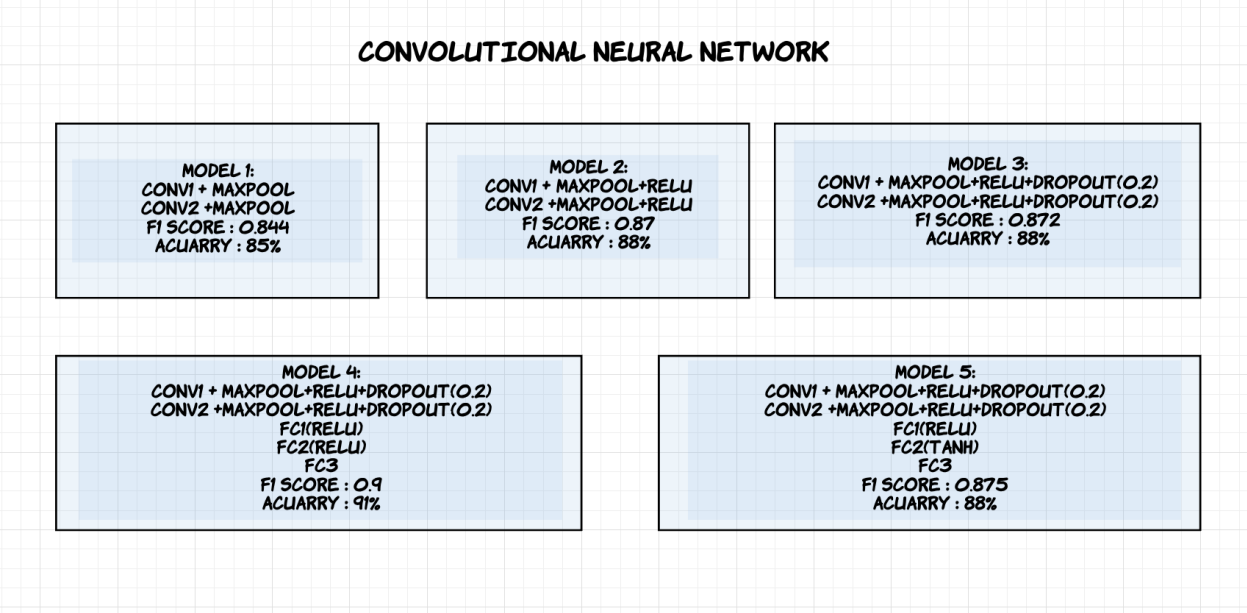

Figure.2: CNN Models

- ***Model 2:*** the second convolution network has almost the same convolution network but adding ReLU activation function on each of feature stages after applying max pooling.
- ***Model 3:*** The third convolution network model has two feature stages, which are the same as model two and trained with dropout applied to each of the two feature stages. The dropout rate applied is 0.2.
- ***Model 4:*** the fourth convolution network model has the same feature stages as model three. In addition, there are 3 fully connected layers. The first fully connected layer has input-feature = 800 and output-feature = 120. The second fully connected layer has input-feature = 120 and output-feature = 84. The third fully connected layer has input-feature = 84 and output-feature = 10. The first and second fully connected layers applied ReLU as well.
- ***Model 5:*** the fifth convolution network model has almost the same model as the fourth model. The normalization module applied on the second fully connected layer change to tanh.
- Besides Pytorch, we also use Caffe on the CNN model 2.

In this project, we also use the mini-batches. Compared with different batch sizes and learning rates, we found out the number 500 for the batch size and 0.001 for the learning rate

works well for our MLP model. Based on the training dataset, we used 531,131 additional training data in order to see there have improvements for our models. Since the extra training data has more clearly photos data, so we easily to see it will improve the accuracy or not. In order to detect and prevent overfitting, we applied from CNN model 3, we found that the accuracy has not improved, which indicates that there is overfitting exist when applying dropout = 0.2. However, when we apply dropout = 0.5, the accuracy decrease, so there is no overfitting.

Results

Based on our results, the convolutional neural network has better performance. We used a two layers MLP, which takes the image pixels vector as input, then the hidden layer with 500 nodes by using ReLU activation function, and finally the output layer with 10 nodes, which are the number of classes in our data. The loss of the MLP model maintains between 0.4 to 0.6. The highest single class accuracy can reach 89% (class 2), and the lowest single accuracy is around 68% (class 3). This simple MLP model can reach the 80% overall accuracy with 10 epochs training process, and the result doesn't change a lot when we are adding more training iterations, try different batch sizes and number of hidden sizes.

	MLP	CNN (ver1)	CNN (ver2)	CNN (ver3)	CNN (ver4)	CNN (ver5)
Accuracy	0.80	0.85	0.88	0.88	0.91	0.88
Time (s)	130.97	142.60	141.79	144.45	141.80	136.55
F1 score	0.79	0.844	0.87	0.872	0.90	0.875

Table.1: CNN & MLP Models Results

	Training dataset	Extra Training dataset
Accuracy	0.91	0.94
Time (s)	141.80	1048.43
F1 score	0.90	0.939

Table.2: CNN Model Four's Results by Different Training dataset

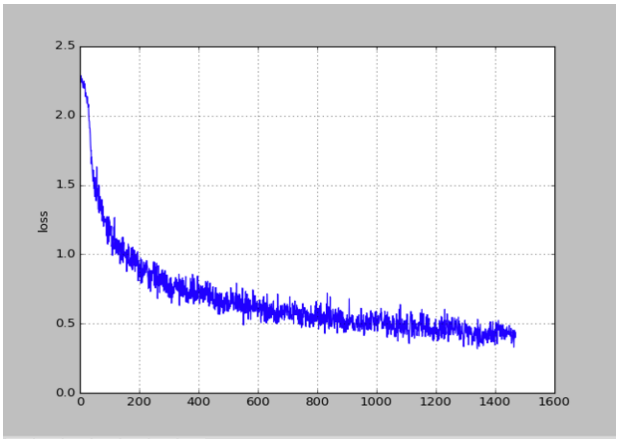

Figure. 3: The loss of MLP

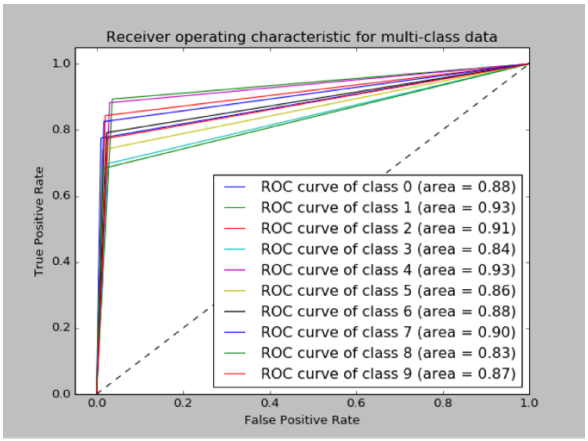

Figure. 4: The ROC of MLP

The most basic CNN model we built contains two convolutional layers with max-pooling layers, which can get the 0.85 accuracy and 0.84 f1-score by 10 epochs training process. We experiment more epochs, but the result does not improve obviously. For the second model, we add the ReLU layers followed each convolutional layer and max-pooling layers. In this one, the f1-score improved to 0.87. Besides, the loss is really similar to the MLP model. The third model we add the dropout layer behind each ReLU layer with 20% dropout ratio. By using the 0.2 dropout rate, the result is close to the previous model, but when we tried 0.5 dropout rate, the performance becomes worse in f1. In model 4, we add three more fully connected layers and the two of them with ReLU functions. This model gets the best performance within those six different models, which gets 91% accuracy and 0.9 f1-score. The training loss bounces and maintains between 0.2 to 0.6, which is better than the results of MLP and previous CNN models. In the last model, we changed the ReLU function to tanh function. The accuracy decreased to

88%, since ReLU function can greatly accelerate the convergence of stochastic gradient descent compared to tanh function.

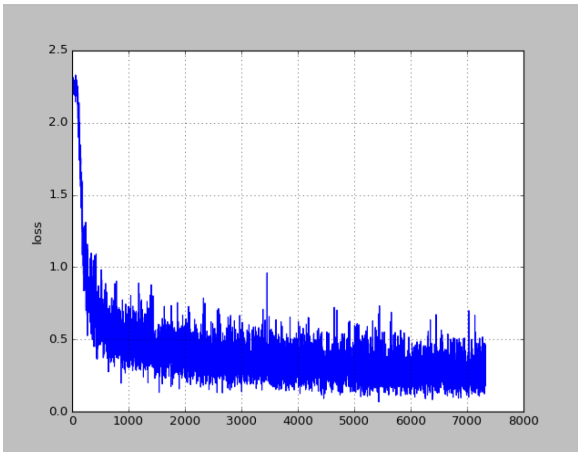

Figure. 5: The loss of CNN Model 4

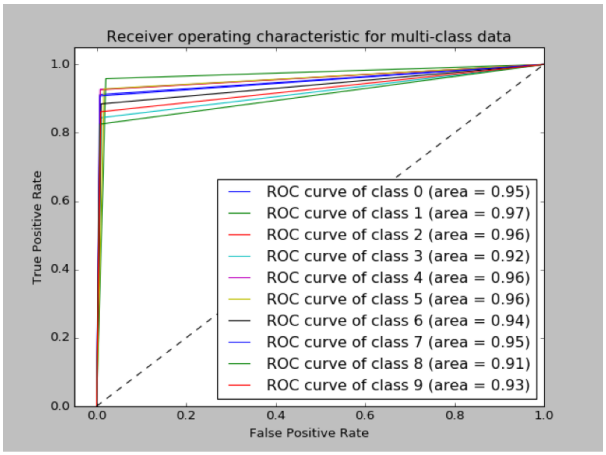

Figure. 6: The ROC of CNN Model 4

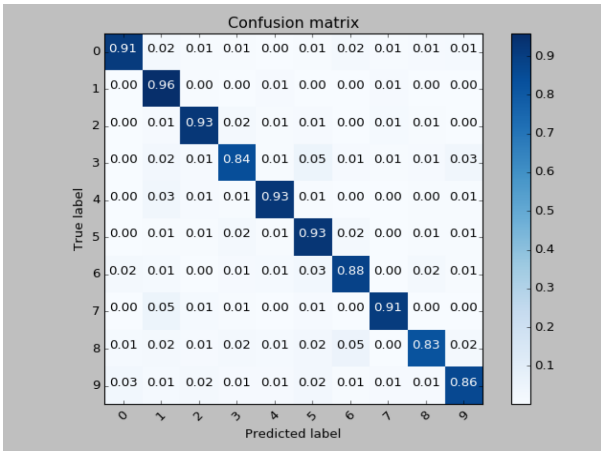

Figure. 7: Confusion Matrix of CNN Model 4

```
Confusion matrix, without normalization
[[1590   31   11   10    3   14   34   13   16   22]
 [  23 4887   24   20   47   16   14   48   15    5]
 [   9   47 3842   67   30   39   13   61   29   12]
 [  10   71   30 2433   16  148   26   23   33   92]
 [   7   80   26   18 2339   13    6    9   12   13]
 [   7   16   13   40   15 2214   47    2   14   16]
 [  39   28    4   17   27   55 1749    9   37   12]
 [   3  107   25   15    4   18    5 1833    3    6]
 [  21   28   15   33   24   41   89    4 1370   35]
 [  47   23   35   18   14   38   11   17   18 1374]]
```

Figure. 8: The Normalized Confusion Matrix Plot of CNN Model 4

Furthermore, we also used the extra dataset as the training data, which contains 531,131 images and more distinct averagely. We tried the MLP model, the first CNN model, and the fourth CNN model by using the extra data as the training set. The fourth CNN model can reach 0.94 f1-score and the loss maintain between 0 to 0.5 which is so much better than the other results, since there are more training data which lowered the occasionality, and also the images are more distinct with the clear and simple background.

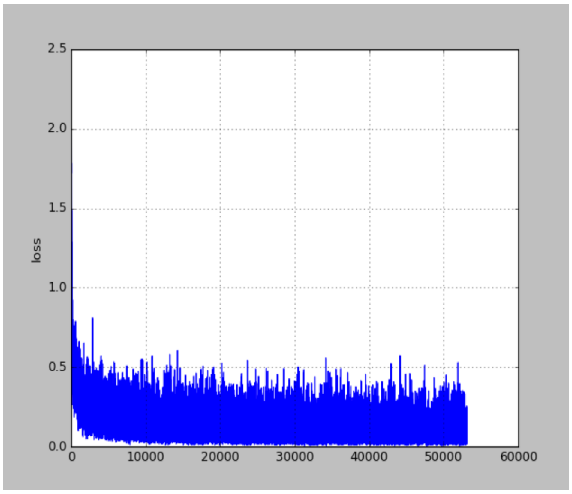

Figure. 9: The Loss of CNN Model 4 by Using Extra Training Data

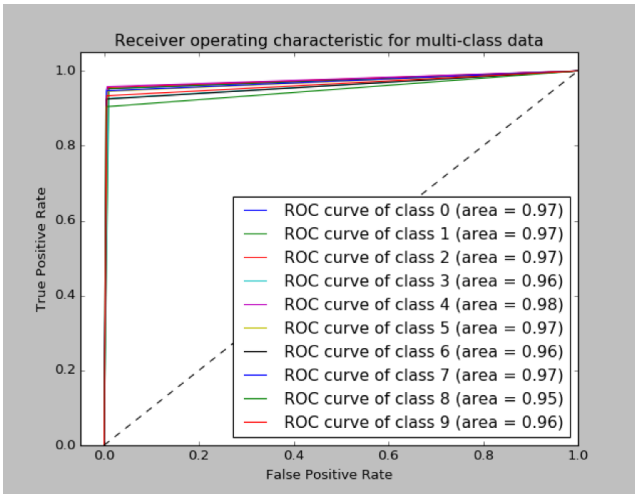

Figure. 10: The ROC of CNN Model 4 by Using Extra Training Data

Otherwise, we transfer the second CNN model in Caffe framework, which gets the same results, but saves tons of training time (70.45s training time in Caffe). We also plot the confusion matrix (both normalized and non-normalized versions) of our results. Each row of the confusion matrix represents the instances in a predicted class while each column represents the instances in the true class. The confusion matrix can easily show if the system mislabeled one as another. From the confusion matrix (and the plot). Since our results are reasonable (79%~94%), so the plot looks like colored-diagonal. Also, we can easily calculate the precision and recall by this matrix. In addition, we plot the ROC curve for each class of our model. The closer the curve follows the left border and the top border of the ROC space, the more accurate the model. The closer the curve comes to the diagonal line of the ROC space, the less accurate the model. The area under the curve is a measure of test accuracy. Compared the ROC plots of our MLP model and the fourth CNN model, the AUC of each class in MLP model is higher than 0.80, and the AUC of each class in CNN is higher than 0.9. The AUC of each class of the forth CNN model by using the extra training data is closer to 1 (higher than 0.95). Thus, the fourth CNN model is more accurate, especially with the very large extra training data.

Conclusion

With training dataset, the state-of-the-art performance on the test dataset with the lowest accuracy of 80% for MLP and the highest accuracy of 91% for CNN model 4. With the same CNN model 4, the performance on the test dataset with training dataset (~ 70k) has the accuracy of 91% and the performance on the test dataset with the extra training dataset (~ 531k) has the accuracy of 94%. The results above show that the CNN has higher accuracy and lower loss than the MLP model. With implementing the same CNN model, the extra training dataset has better outcomes than the training dataset, because the extra training dataset has simpler and less blurry images, which means the digits in the image will be easier to be detected. Because the training dataset contains some of the images that are even harder to recognize at the first glance. It also comes from a significantly harder real-world problem of recognizing digits and numbers in natural scene images.

The other difficult part of this project is to decide the proper hyper-parameters such as the learning rate. In the scholar research, a case with the application of advanced optimization techniques and applying 7 layers that achieved over 93% accuracy. A comprehensive review of these examples would be meaningful future work to consider. In addition, when comparing the processing of the MLP model and multiple CNN models, we have learned that the running time of the CNN training model will be shorter when compared to the MLP model. The CNN model can reach the same performance (80% accuracy) by using only 14s (One epoch training of our CNN model). Even though the multilayer network can work on lots of input sources, convolution network works better on images. In our project, since training and testing datasets are made of images, the convolution network will be better. Even more, the convolution network has weighed sharing, so the processing time will be shorter than that of the multilayer network.

A couple of interesting next steps to be pursued further research. For example, from the research "*Generalizing Pooling Functions in Convolutional Neural Networks*", they find that “mixed” (with one mix per pooling layer) is outperformed by “gated” with one gate per pooling layer. Changing the different Pooling layers may change the results for our relatively large SVHN dataset. We are also thinking if we add more layer for our network, maybe we can improve our results too.

References

Lee, Chen-Yu. “Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree.” *Eprint ArXiv*, 10 Oct. 2015.

Netzer, Yuval et al. “Reading Digits in Natural Images with Unsupervised Feature Learning.” (2011).

Sermanet, Pierre et al. “Convolutional neural networks applied to house numbers digit classification.” *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)* (2012): 3288-3291.

Appendix: Codes

MLP (MLP.py):

```
import torch
import torchvision
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import itertools
import numpy as np
import scipy.io as sio
import time

#################################################################################################################
# Hyper Parameters
input_size = 3*32*32
hidden_size = 500
num_classes = 10
num_epochs = 10
batch_size = 500
learning_rate = 0.001

#################################################################################################################
train_dataset = dsets.SVHN(root='./data_svhn', split='train', download=True,
                           transform=transforms.Compose([
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]))

test_dataset = dsets.SVHN(root='./data_svhn', split='test', download=True,
                          transform=transforms.Compose([
                              transforms.ToTensor(),
                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]))

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

classes = ['0','1','2','3','4','5','6','7','8','9']

#################################################################################################################
# Plot of data
mat=sio.loadmat('data_svhn/train_32x32.mat')
data=mat['X']
label=mat['y']
for i in range(20):
    plt.subplot(4,5,i+1)
```

```
for i in range(20):
    plt.subplot(4,5,i+1)
    plt.title(label[i][0])
    plt.imshow(data[...,i])
    plt.axis('off')
plt.show()

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

dataiter = iter(train_loader)
images, labels = dataiter.next()
imshow(torchvision.utils.make_grid(images))
plt.show()

print(' '.join('%5s' % classes[labels[j]] for j in range(8)))

#########################################################################################################


# Neural Network Model (1 hidden layer)
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

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


#########################################################################################################
net = Net(input_size, hidden_size, num_classes)
net.cuda()

#########################################################################################################
# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
#########################################################################################################

loss_list = []

start = time.time()
```

imshow()

```
# Train the Model
for epoch in range(num_epochs):
    for i, data in enumerate(train_loader):

        images, labels = data
        images, labels = Variable(images.cuda()), Variable(labels.cuda())
        optimizer.zero_grad()
        outputs = net(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'
                  % (epoch + 1, num_epochs, i + 1, len(train_dataset) // batch_size, loss.data[0]))

end = time.time()
total_time = end - start

#########################################################################################################
# Test the Model
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images.view(-1, 3* 32 * 32)).cuda()
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted.cpu() == labels.cpu()).sum()

print('Accuracy of the network on the 26032 test images: %d %%' % (100 * correct / total))

#########################################################################################################

_, predicted = torch.max(outputs.data, 1)

print('Predicted label: ', ' '.join('%5s' % predicted[j].cpu().numpy() for j in range(20)))
print('Actual label: ', ' '.join('%5s' % labels[j].cpu().numpy() for j in range(20)))

#########################################################################################################

# There is bug here find it and fix it
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

test_labels_list = []
test_predicted_list = []

all_test_labels = []
all_test_predicted = []
```

imshow()

```
for images, labels in test_loader:
    images = Variable(images.view(-1, 3 * 32 * 32)).cuda()
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    labels = labels.cpu().numpy()
    c = (predicted.cpu().numpy() == labels)

    for i in range(len(labels)):
        test_labels_list.append(classes[labels[i]])
        test_predicted_list.append(classes[predicted.cpu()[i]])
        all_test_labels.append(labels[i])
        all_test_predicted.append(predicted.cpu()[i])

    for i in range(len(labels)):
        label = labels[i]
        class_correct[label] += c[i]
        class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] / class_total[i]))

from sklearn.metrics import (precision_score, recall_score, f1_score)
from sklearn.metrics import classification_report

print('Classification report:\n', classification_report(test_labels_list, test_predicted_list))

print("Precision: %1.3f" % precision_score(test_labels_list, test_predicted_list, average='macro'))
print("Recall: %1.3f" % recall_score(test_labels_list, test_predicted_list, average='macro'))
print("F1: %1.3f\n" % f1_score(test_labels_list, test_predicted_list, average='macro'))

print('Training time:', total_time)

# Save the Model
torch.save(net.state_dict(), 'model.pkl')

################################################################################################################

# Plot loss
plt.plot(loss_list)
plt.grid(True, which='both')
plt.ylabel('loss')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.show()

################################################################################################################

# Plot ROC curve
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
y_label = label_binarize(all_test_labels, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
y_predict = label_binarize(all_test_predicted, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(y_label[:, i], y_predict[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

##########################################################################################################

for i in range(len(classes)):
    plt.plot(fpr[i], tpr[i],
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()

##########################################################################################################

def confusion_matrix_plot(y_test, y_pred, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    cm = confusion_matrix(y_test, y_pred)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()


confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=False)

confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=True)
```

CNN Model 4 (CNN4.py):

```
import torch
import torch.nn as nn
import torchvision
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import time
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import itertools
import torch.nn.functional as F
import numpy as np

# ##############################################################################################################
# # Hyper Parameters
num_epochs = 10
batch_size = 100
learning_rate = 0.001
# ##############################################################################################################

train_dataset = dsets.SVHN(root='./data_svhn', split='train', download=True,
                           transform=transforms.Compose([
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))

test_dataset = dsets.SVHN(root='./data_svhn', split='test', download=True,
                          transform=transforms.Compose([
                              transforms.ToTensor(),
                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                          ]))

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']


# ##############################################################################################################

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


dataiter = iter(train_loader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images))
plt.show()
```

```
print(' '.join('%5s' % classes[labels[j]] for j in range(8)))


########################################################################################################################

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.dropout(x)
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = x.view(-1, 32 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


cnn = CNN()
cnn.cuda()
########################################################################################################################

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
########################################################################################################################

loss_list = []
start = time.time()
# Train the Model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()

        # Forward + Backward + Optimize
        optimizer.zero_grad()
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())
```

```
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f'
                  % (epoch + 1, num_epochs, i + 1, len(train_dataset) // batch_size, loss.item()))

end = time.time()
total_time = end - start
###############################################################################################################

# Test the Model
cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted.cpu() == labels).sum()

print('Test Accuracy of the model on the 26032 test images: %d %%' % (100 * correct / total))

###############################################################################################################
# Save the Trained Model
torch.save(cnn.state_dict(), 'cnn.pkl')

###############################################################################################################

_, predicted = torch.max(outputs.data, 1)
print('Predicted label: ', ' '.join('%5s' % predicted[j].cpu().numpy() for j in range(20)))
print('Actual label: ', ' '.join('%5s' % labels[j].cpu().numpy() for j in range(20)))

###############################################################################################################

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

test_labels_list = []
test_predicted_list = []

all_test_labels = []
all_test_predicted = []

for images, labels in test_loader:
    images = Variable(images).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
```

```
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    labels = labels.cpu().numpy()
    c = (predicted.cpu().numpy() == labels)

    for i in range(len(labels)):
        test_labels_list.append(classes[labels[i]])
        test_predicted_list.append(classes[predicted.cpu()[i]])
        all_test_labels.append(labels[i])
        all_test_predicted.append(predicted.cpu()[i])

    for i in range(len(labels)):
        label = labels[i]
        class_correct[label] += c[i]
        class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] / class_total[i]))

#########################################################################################################

# Save the Trained Model
torch.save(cnn.state_dict(), 'cnn.pkl')

from sklearn.metrics import (precision_score, recall_score, f1_score)
from sklearn.metrics import classification_report

print('Classification report:\n', classification_report(test_labels_list, test_predicted_list))

print("Precision: %1.3f" % precision_score(test_labels_list, test_predicted_list, average='macro'))
print("Recall: %1.3f" % recall_score(test_labels_list, test_predicted_list, average='macro'))
print("F1: %1.3f\n" % f1_score(test_labels_list, test_predicted_list, average='macro'))

print('Training time:', total_time)

#########################################################################################################

plt.plot(loss_list)
plt.grid(True, which='both')
plt.ylabel('loss')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.show()

#########################################################################################################

# Plot ROC curve
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
y_label = label_binarize(all_test_labels, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
y_predict = label_binarize(all_test_predicted, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
fpr = dict()
```

```
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
    fpr[i], tpr[i], _ = roc_curve(y_label[:, i], y_predict[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

#########################################################################################################

for i in range(len(classes)):
    plt.plot(fpr[i], tpr[i],
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()

#########################################################################################################
```

```
#########################################################################################################################

def confusion_matrix_plot(y_test, y_pred, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    cm = confusion_matrix(y_test, y_pred)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()


confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=False)

confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=True)
```

CNN Model 4: Use Extra Training Data (CNN4_extra.py)

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
import time
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import itertools
import torch.nn.functional as F
import numpy as np

########################################################################################################################
# Hyper Parameters
num_epochs = 10
batch_size = 100
learning_rate = 0.001
########################################################################################################################

train_dataset = dsets.SVHN(root='./data_svhn', split='extra', download=True,
                           transform=transforms.Compose([
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))

test_dataset = dsets.SVHN(root='./data_svhn', split='test', download=True,
                          transform=transforms.Compose([
                              transforms.ToTensor(),
                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                          ]))

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']


########################################################################################################################

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


dataiter = iter(train_loader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images))
plt.show()
```

```
print(' '.join('%5s' % classes[labels[j]] for j in range(8)))


########################################################################################################################

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.2)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.dropout(x)
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.dropout(x)
        x = x.view(-1, 32 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


cnn = CNN()
cnn.cuda()
########################################################################################################################

# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=learning_rate)
########################################################################################################################

loss_list = []
start = time.time()
# Train the Model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = Variable(images).cuda()
        labels = Variable(labels).cuda()

        # Forward + Backward + Optimize
        optimizer.zero_grad()
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())
```

```
        loss_list.append(loss.item())
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f'
                  % (epoch + 1, num_epochs, i + 1, len(train_dataset) // batch_size, loss.item()))

end = time.time()
total_time = end - start
################################################################################################################

# Test the Model
cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).
correct = 0
total = 0
for images, labels in test_loader:
    images = Variable(images).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted.cpu() == labels).sum()

print('Test Accuracy of the model on the 26032 test images: %d %%' % (100 * correct / total))

################################################################################################################
# Save the Trained Model
torch.save(cnn.state_dict(), 'cnn.pkl')

################################################################################################################

_, predicted = torch.max(outputs.data, 1)
print('Predicted label: ', ' '.join('%5s' % predicted[j].cpu().numpy() for j in range(20)))
print('Actual label: ', ' '.join('%5s' % labels[j].cpu().numpy() for j in range(20)))

################################################################################################################

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

test_labels_list = []
test_predicted_list = []

all_test_labels = []
all_test_predicted = []

for images, labels in test_loader:
    images = Variable(images).cuda()
    outputs = cnn(images)
    _, predicted = torch.max(outputs.data, 1)
    labels = labels.cpu().numpy()
    c = (predicted.cpu().numpy() == labels)
```

```
    c = (predicted.cpu().numpy() == labels)

    for i in range(len(labels)):
        test_labels_list.append(classes[labels[i]])
        test_predicted_list.append(classes[predicted.cpu()[i]])
        all_test_labels.append(labels[i])
        all_test_predicted.append(predicted.cpu()[i])

    for i in range(len(labels)):
        label = labels[i]
        class_correct[label] += c[i]
        class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] / class_total[i]))

#######################################################################################################

# Save the Trained Model
torch.save(cnn.state_dict(), 'cnn.pkl')

from sklearn.metrics import (precision_score, recall_score, f1_score)
from sklearn.metrics import classification_report

print('Classification report:\n', classification_report(test_labels_list, test_predicted_list))

print("Precision: %1.3f" % precision_score(test_labels_list, test_predicted_list, average='macro'))
print("Recall: %1.3f" % recall_score(test_labels_list, test_predicted_list, average='macro'))
print("F1: %1.3f\n" % f1_score(test_labels_list, test_predicted_list, average='macro'))

print('Training time:', total_time)

#######################################################################################################

plt.plot(loss_list)
plt.grid(True, which='both')
plt.ylabel('loss')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.show()

#######################################################################################################

# Plot ROC curve
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
y_label = label_binarize(all_test_labels, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
y_predict = label_binarize(all_test_predicted, classes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(classes)):
```

```
    fpr[i], tpr[i], _ = roc_curve(y_label[:, i], y_predict[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

##########################################################################################################

for i in range(len(classes)):
    plt.plot(fpr[i], tpr[i],
             label='ROC curve of class {0} (area = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()

##########################################################################################################

def confusion_matrix_plot(y_test, y_pred, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):
    cm = confusion_matrix(y_test, y_pred)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()


confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=False)
```

```
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
    plt.show()


confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=False)

confusion_matrix_plot(test_labels_list, test_predicted_list, classes, normalize=True)
```

Caffe (mlcnn_sol.py)

```python
########################################CREATE TRAINING#################################################################
import os
import caffe
import matplotlib.pyplot as plt
import numpy as np
import numpy
import time

#####################################Train the Network with the Solver#########################################

caffe.set_mode_gpu()

solver = caffe.SGDSolver('mlcnn_solver.prototxt')


start = time.time()

niter =20000
test_interval = 100
train_loss = np.zeros(niter)
test_acc = np.zeros(int(np.ceil(niter / test_interval)))

# the main solver loop
for it in range(niter):
    solver.step(1)  # SGD by Caffe

    # store the train loss
    train_loss[it] = solver.net.blobs['loss'].data
    solver.test_nets[0].forward(start='conv1')

    if it % test_interval == 0:
        acc=solver.test_nets[0].blobs['accuracy'].data
        print 'Iteration', it, 'testing...','accuracy:',acc
        test_acc[it // test_interval] = acc

end = time.time()
total_time = end-start



#-------------------------------------------------------------------------------------------------------
########################Plotting Intermediate Layers, Weight#################################
#-------------------------------------Define Functions-------------------------------------------------

def vis_square_f(data):
    """Take an array of shape (n, height, width) or (n, height, width, 3)
       and visualize each (height, width) thing in a grid of size approx. sqrt(n) by sqrt(n)"""

    # normalize data for display
    #data = (data - data.min()) / (data.max() - data.min())
    data -= data.min()
    data /= data.max()

    # force the number of filters to be square
    n = int(np.ceil(np.sqrt(data.shape[0])))
    padding = (((0, n ** 2 - data.shape[0]),
                (0, 1), (0, 1))  # add some space between filters
               + ((0, 0),) * (data.ndim - 3))  # don't pad the last dimension (if there is one)
    data = np.pad(data, padding, mode='constant', constant_values=(0,0))  # pad with ones (white)

    # tile the filters into an image
    data = data.reshape((n, n) + data.shape[1:]).transpose((0, 2, 1, 3) + tuple(range(4, data.ndim + 1)))
```

```
        data = data.reshape((n * data.shape[1], n * data.shape[3]) + data.shape[4:])
        plt.figure()
        #plt.imshow(data,cmap='Greys',interpolation='nearest')
        plt.imshow(data)#, cmap='gray')
        plt.axis('off')
    #-------------------------------------------------------------------------------------------------------
    #-------------------------------Plot All Feature maps Functions-----------------------------------------
    plt.figure(1)
    plt.semilogy(np.arange(niter), train_loss)
    plt.xlabel('Number of Iteration')
    plt.ylabel('Training Loss Values')
    plt.title('Training Loss')
    plt.show()

    plt.figure(2)
    plt.plot(test_interval * np.arange(len(test_acc)), test_acc, 'r')
    plt.xlabel('Number of Iteration')
    plt.ylabel('Test Accuracy Values')
    plt.title('Test Accuracy')
    plt.show()
    #-------------------------------------------------------------------------------------------------------
    #-------------------------------Plot All Feature maps Functions-----------------------------------------
    net = solver.net
    f1_0 = net.blobs['conv1'].data[0]
    vis_square_f(f1_0)
    plt.title('Feature Maps for Conv1')
    plt.show()

    vis_square_f(net.blobs['pool1'].data[0])
    plt.show()

    #-------------------------------------------------------------------------------------------------------
    #----------------------------Print Shape ans Sizes for all Layers---------------------------------------

    for layer_name, blob in net.blobs.iteritems():
        print layer_name + '\t' + str(blob.data.shape)

    for layer_name, param in net.params.iteritems():
        print layer_name + '\t' + str(param[0].data.shape), str(param[1].data.shape)

    print('Time:',total_time)
```

mlcnn_solver.prototxt

```
net: "mlcnn_train_test.prototxt"
test_iter: 100
# Carry out testing every 1000 training data
test_interval: 1000
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.001
momentum: 0.9
weight_decay: 0.004
# The learning rate policy
lr_policy: "multistep"
gamma:0.1
stepvalue:60000
stepvalue:65000
stepvalue:70000
# Display every 200 iterations
display: 200
# The maximum number of iterations
max_iter: 70000
# snapshot intermediate results
snapshot: 10000
snapshot_prefix: "cifar10_full"
# solver mode: CPU or GPU
solver_mode: GPU
```

mlcnn_train_test.prototxt

mlcnn_train_test.prototxt

```
name: "SVHN"
layer {
  name: "svhn"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mean_file: "mean.binaryproto"
  }
  data_param {
    source: "svhn_train_lmdb"
    batch_size: 50
    backend: LMDB
  }
}
layer {
  name: "svhn"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    mean_file: "mean.binaryproto"
  }
  data_param {
    source: "svhn_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 16
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.0001
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "pool1"
  top: "pool1"
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 32
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
```

```
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "pool2"
  top: "pool2"
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  inner_product_param {
    num_output: 120
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip1"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip1"
  bottom: "label"
  top: "loss"
}
```

caffe_covert_lmdb.py

```python
import numpy as np
import caffe
import lmdb
import scipy.io as sio
import random
from caffe.proto import caffe_pb2

def main():
    train=sio.loadmat('data_svhn/train_32x32.mat')
    test=sio.loadmat('data_svhn/test_32x32.mat')

    train_data=train['X']
    train_label=train['y']
    test_data=test['X']
    test_label=test['y']

    train_data = np.swapaxes(train_data, 0, 3)
    train_data = np.swapaxes(train_data, 1, 2)
    train_data = np.swapaxes(train_data, 2, 3)

    test_data = np.swapaxes(test_data, 0, 3)
    test_data = np.swapaxes(test_data, 1, 2)
    test_data = np.swapaxes(test_data, 2, 3)

    N=train_label.shape[0]
    map_size=train_data.nbytes*10
    env=lmdb.open('svhn_train_lmdb',map_size=map_size)
    txn=env.begin(write=True)

    #shuffle the training data
    r=list(range(N))
    random.shuffle(r)

    count=0
    for i in r:
        datum=caffe_pb2.Datum()
        label=int(train_label[i][0])
        if label==10:
            label=0
        datum=caffe.io.array_to_datum(train_data[i],label)
        str_id='{:08}'.format(count)
        txn.put(str_id,datum.SerializeToString())

        count += 1
        if count % 1000 == 0:
            print('already handled with {} pictures'.format(count))
            txn.commit()
            txn = env.begin(write=True)

    txn.commit()
    env.close()

    map_size = test_data.nbytes * 10
    env = lmdb.open('svhn_test_lmdb', map_size=map_size)
    txn = env.begin(write=True)
    count = 0
    for i in range(test_label.shape[0]):
        datum = caffe_pb2.Datum()
        label = int(test_label[i][0])
        if label == 10:
            label = 0
        datum = caffe.io.array_to_datum(test_data[i], label)
        str_id = '{:08}'.format(count)
        txn.put(str_id, datum.SerializeToString())

        count += 1
        if count % 1000 == 0:
            print('already handled with {} pictures'.format(count))
            txn.commit()
            txn = env.begin(write=True)

    txn.commit()
    env.close()

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