Convolutional Network Assignment

April 16, 2019

0.1 Neural Network Assignment: Image Classification on MNIST

In this assignment, you will design and implement first a multi-layer perceptron model and then a convolutional network model in Pytorch. We will use the CIFAR-10 image dataset. Please complete the rest of the notebook by doing the following tasks.

- Build your network (A MLP model and a CONVNET model). There are some requirements on the network structure, please check the corresponding code sections.
- Train your network.
- Describe your training procedure. Plot the following:
    1. Training and validation loss vs. training iterations.
    2. Training and validation accuracy vs. training iterations.
- Report a final test result on 10000 separate testing examples.
- Give detailed explanation of your code

You can follow the skeleton code in the notebook to proceed. Look for the ##TODO## mark, that's where you should write your code.

Check Pytorch documentation for reference: https://pytorch.org/tutorials/

0.2 1.1 Download data

From this website: https://www.cs.toronto.edu/~kriz/cifar.html Download the CIFAR-10 python version, which is 163 MB. Extract the file you downloaded, you should be able to see a folder named cifar-10-batches-py. The data files are inside. ('data_batch_1', 'data_batch_2', etc..). To make things easier put these data files in a new folder called 'cifar-10-data'. The folder 'cifar-10-data' should be in the same directory as your jupyter notebook. Now we load the data into memory.

1 TODO you should download data

1.1 1.2 Loading Data: you should read the website's data description and have a better understanding of the data (quote):

Loaded in this way, each of the batch files contains a dictionary with the following elements: data -- a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red

channel values of the first row of the image. labels -- a list of 10000 numbers in the range 0-9. The number at index i indicates the label of the ith image in the array data.

The dataset contains another file, called batches.meta. It too contains a Python dictionary object. It has the following entries: label_names -- a 10-element list which gives meaningful names to the numeric labels in the labels array described above. For example, label_names[0] == "airplane", label_names[1] == "automobile", etc.

```
In [1]: import numpy as np
        import torch
        import torch.nn as nn
        import torch.optim as optim
        import torch.nn.functional as F

In [2]: ## follow the instructions on the website
        def unpickle(file):
            ## used to read binary files since our data files are in binary format
            import pickle
            with open(file, 'rb') as fo:
                dict = pickle.load(fo, encoding='bytes')
            return dict

In [3]: ## loading data from binary data files
        batch_1_dictionary = unpickle('cifar-10-data/data_batch_1')
        batch_2_dictionary = unpickle('cifar-10-data/data_batch_2')

In [4]: batch_1_dictionary.keys()

Out[4]: dict_keys([b'batch_label', b'labels', b'data', b'filenames'])

In [6]: ## get training, validation and testing sets
        X_train_all = np.array(batch_1_dictionary[b'data']).reshape(10000,3,32,32)
        y_train_all = np.array(batch_1_dictionary[b'labels'])
        validation_count = 1000
        train_count = X_train_all.shape[0]-1000
        X_train = X_train_all[:train_count]
        y_train = y_train_all[:train_count]
        X_val = X_train_all[train_count:]
        y_val = y_train_all[train_count:]
        X_test = np.array(batch_2_dictionary[b'data']).reshape(10000,3,32,32)
        y_test = np.array(batch_2_dictionary[b'labels'])

In [7]: X_train.shape, y_train.shape, X_val.shape, y_val.shape, X_test.shape, y_test.shape

Out[7]: ((9000, 3, 32, 32),
         (9000,),
         (1000, 3, 32, 32),
         (1000,),
         (10000, 3, 32, 32),
         (10000,))
```

1.2 1.3 Preprocess Data

Typically public datasets have some kind of data description or manual that we can use to gain basic understanding of the dataset. You should always try to look for such things and utilize them before you start processing the data. We now want to preprocess the data.

We now want to do a data normalization. (Do you still remember why data normalization is important? Refer to Normalizing Input in first week of Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization) You can typically just do the minus by mean and divide by std normalization. But since we are dealing with image data, we can use another type of data normalization: simply divide by 255 to bring all features to the range (0,1). (Think about why the number 255?) There are different ways to normalize data, feel free to try more options if you are interested.

2 TODO

write your data normalization function here

```
In [7]: # for RGB data we can simply divide by 255
        X_train_normalized =
        X_val_normalized =
        X_test_normalized =
```

2.1 1.4 Take a look at the data

We can plot the image data to have a better understanding of it.

```
In [8]: ## class label related
        CLASSES = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru

        def index_to_class_name(y):
            return CLASSES[y]
        def class_name_to_index(class_name):
            return CLASSES.index(class_name)
```

```
In [9]: import math
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg
        """
        Plotting utilities, if you want to know how these work exactly, check the reference
        Or the documentations
        reference:
        https://matplotlib.org/users/image_tutorial.html
        https://stackoverflow.com/questions/46615554/how-to-display-multiple-images-in-one-fig
        """

        def show_single_image(data):
            # show a single image
            ## note that using matplotlib plotting function, we will have to reshape the data
```

```
        img = data.reshape(3,32,32).transpose(1,2,0)
        imgplot = plt.imshow(img)
    def show_multiple_images(data, data_y, n_show=12, columns=4):
        ## given an array of data, show all of them as images
        fig=plt.figure(figsize=(8, 8))
        n = min(data.shape[0], n_show)
        rows = math.ceil(n/columns)
        for i in range(n):
            img = data[i].reshape(3,32,32).transpose(1,2,0)
            ax = fig.add_subplot(rows, columns, i+1) ## subplot index starts from 1 not 0
            class_name = index_to_class_name(data_y[i])
            ax.set_title(str(data_y[i])+": "+class_name)
            plt.imshow(img)
        plt.show()
```

In [10]:
```
show_multiple_images(X_train, y_train)
```

2.2 2.1 Design Network Structure of a MLP Model

Reference: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

When we use a multi-layer perceptron (a very basic neural net with fully connected layers), we will need to flatten the data so that the MLP can use it. Each data point in our training data is currently in the form of 1x32x32x3, we want to convert it into 1x3072, so that its shape makes sense to our MLP's input layer. Recall that in our MLP, the input size for the first layer is 3072. There are multiple ways of doing this, we can directly flatten all data points, or we can also do the flattening for each mini-batch.

In terms of the loss function, since we are doing the task of multi-class classification, we will use the cross entropy loss.

We will use nn.CrossEntropyLoss to compute this, refer to Pytorch doc for details. A short summary is "this criterion combines nn.LogSoftmax() and nn.NLLLoss() in one single class" That basically means we don't have to worry about softmax layer and the log part. Make sure you read the Pytorch doc to understand exactly how it works.

3 TODO

write your network structure here

1. you should only use linear layers (nn.Linear), should use 2 or more linear layers
2. and you should use relu as activation
3. after the final layer we DON'T HAVE activation
4. you should not use softmax here because pytorch's BCE loss includes softmax
5. for hidden layer size, anything between 32-256 is OK

```
In [11]: import torch
         import torch.nn as nn
         import torch.nn.functional as F
         from torch import Tensor
         import torch.optim as optim

         class MLP(nn.Module):
             ## a very simple MLP model
             def __init__(self, input_dim, output_dim):
                 super(MLP, self).__init__()
                 self.fc1 =
                 self.fc2 =
             def forward(self, x):

                 return
```

```
In [12]: # utility for getting prediction accuracy
         def get_correct_and_accuracy(y_pred, y):
             # y_pred is the nxC prediction scores
             # give the number of correct and the accuracy
             n = y.shape[0]
             # find the prediction class label
```

```
    _ ,pred_class = y_pred.max(dim=1)
    correct = (pred_class == y).sum().item()
    return correct ,correct/n
```

3.1 2.2 Training the MLP model

1. We first initialize the neural network, and also the optimizer and the criterion (loss function)
2. We then convert data into correct tensor form (note for MLP we need to flatten the image data, from 3x32x32 to 3072)
3. We then start doing training iterations

4 TODO

write code here 1. init your model 2. use Adam optimizer, with lr=1e-3 3. use cross entropy criterion 4. for X_train, X_val, cast to Tensor and then reshape them to flatten them (change shape to mx3072) 5. for y_train, y_val, cast to Tensor and then cast to type long 6. simply do 100 iterations

```
In [15]: # init network
         mlp = MLP(3072, 10)
         print('model structure: ',mlp)
         # init optimizer
         optimizer =
         # set loss function
         criterion =

         # prepare for mini-batch stochastic gradient descent
         n_iteration = 100
         batch_size = 256
         n_train_data = X_train_normalized.shape[0]
         n_batch = int(np.ceil(n_train_data/batch_size))

         # convert X_train and X_val to tensor and flatten them
         X_train_tensor =
         X_val_tensor =

         # convert training label to tensor and to type long
         y_train_tensor =
         y_val_tensor =
```

```
model structure:  MLP(
  (fc1): Linear(in_features=3072, out_features=32, bias=True)
  (fc2): Linear(in_features=32, out_features=10, bias=True)
)
```

5 TODO

1. Here you will have to write training code

2. As well as code to log the training process for later plotting
3. After each iteration through training data, you should use that print function to log you current training process

```
In [16]: ## start
         train_loss_list = np.zeros(n_iteration)
         train_accu_list = np.zeros(n_iteration)
         val_loss_list = np.zeros(n_iteration)
         val_accu_list = np.zeros(n_iteration)
         for i in range(n_iteration):
             # first get a minibatch of data
             for j in range(n_batch):
                 batch_start_index = j*batch_size
                 # get data batch from the normalized data
                 X_batch = X_train_tensor[batch_start_index:batch_start_index+batch_size]
                 # get ground truth label y
                 y_batch = y_train_tensor[batch_start_index:batch_start_index+batch_size]


             print("Iter %d ,Train loss: %.3f, Train acc: %.3f, Val loss: %.3f, Val acc: %.3f"
                   %(i ,ave_train_loss, train_accu, val_loss, val_accu))

             ## add to the logs so that we can use them later for plotting
             train_loss_list[i] =
             train_accu_list[i] =
             val_loss_list[i] =
             val_accu_list[i] =
```

```
Iter 0 ,Train loss: 2.195, Train acc: 0.188, Val loss: 2.064, Val acc: 0.253
Iter 1 ,Train loss: 2.009, Train acc: 0.279, Val loss: 1.959, Val acc: 0.297
Iter 2 ,Train loss: 1.922, Train acc: 0.316, Val loss: 1.915, Val acc: 0.305
Iter 3 ,Train loss: 1.873, Train acc: 0.336, Val loss: 1.885, Val acc: 0.329
Iter 4 ,Train loss: 1.837, Train acc: 0.353, Val loss: 1.864, Val acc: 0.335
Iter 5 ,Train loss: 1.812, Train acc: 0.359, Val loss: 1.850, Val acc: 0.350
Iter 6 ,Train loss: 1.788, Train acc: 0.369, Val loss: 1.835, Val acc: 0.354
Iter 7 ,Train loss: 1.766, Train acc: 0.375, Val loss: 1.822, Val acc: 0.361
Iter 8 ,Train loss: 1.750, Train acc: 0.381, Val loss: 1.813, Val acc: 0.359
Iter 9 ,Train loss: 1.734, Train acc: 0.386, Val loss: 1.804, Val acc: 0.366
Iter 10 ,Train loss: 1.721, Train acc: 0.392, Val loss: 1.795, Val acc: 0.367
Iter 11 ,Train loss: 1.708, Train acc: 0.397, Val loss: 1.788, Val acc: 0.373
Iter 12 ,Train loss: 1.697, Train acc: 0.399, Val loss: 1.786, Val acc: 0.376
Iter 13 ,Train loss: 1.688, Train acc: 0.403, Val loss: 1.782, Val acc: 0.375
Iter 14 ,Train loss: 1.678, Train acc: 0.407, Val loss: 1.778, Val acc: 0.372
Iter 15 ,Train loss: 1.668, Train acc: 0.413, Val loss: 1.774, Val acc: 0.378
Iter 16 ,Train loss: 1.660, Train acc: 0.415, Val loss: 1.767, Val acc: 0.379
Iter 17 ,Train loss: 1.651, Train acc: 0.418, Val loss: 1.763, Val acc: 0.380
Iter 18 ,Train loss: 1.644, Train acc: 0.420, Val loss: 1.761, Val acc: 0.377
```

```
Iter 19 ,Train loss: 1.637, Train acc: 0.423, Val loss: 1.758, Val acc: 0.378
Iter 20 ,Train loss: 1.630, Train acc: 0.426, Val loss: 1.756, Val acc: 0.380
Iter 21 ,Train loss: 1.623, Train acc: 0.431, Val loss: 1.755, Val acc: 0.376
Iter 22 ,Train loss: 1.617, Train acc: 0.433, Val loss: 1.754, Val acc: 0.370
Iter 23 ,Train loss: 1.612, Train acc: 0.433, Val loss: 1.752, Val acc: 0.376
Iter 24 ,Train loss: 1.606, Train acc: 0.434, Val loss: 1.754, Val acc: 0.378
Iter 25 ,Train loss: 1.600, Train acc: 0.436, Val loss: 1.754, Val acc: 0.380
Iter 26 ,Train loss: 1.596, Train acc: 0.437, Val loss: 1.753, Val acc: 0.381
Iter 27 ,Train loss: 1.590, Train acc: 0.439, Val loss: 1.752, Val acc: 0.381
Iter 28 ,Train loss: 1.586, Train acc: 0.440, Val loss: 1.748, Val acc: 0.388
Iter 29 ,Train loss: 1.580, Train acc: 0.442, Val loss: 1.746, Val acc: 0.385
Iter 30 ,Train loss: 1.576, Train acc: 0.445, Val loss: 1.746, Val acc: 0.383
Iter 31 ,Train loss: 1.572, Train acc: 0.446, Val loss: 1.742, Val acc: 0.385
Iter 32 ,Train loss: 1.568, Train acc: 0.447, Val loss: 1.742, Val acc: 0.386
Iter 33 ,Train loss: 1.563, Train acc: 0.449, Val loss: 1.740, Val acc: 0.383
Iter 34 ,Train loss: 1.559, Train acc: 0.451, Val loss: 1.735, Val acc: 0.385
Iter 35 ,Train loss: 1.556, Train acc: 0.454, Val loss: 1.731, Val acc: 0.389
Iter 36 ,Train loss: 1.550, Train acc: 0.453, Val loss: 1.730, Val acc: 0.392
Iter 37 ,Train loss: 1.546, Train acc: 0.455, Val loss: 1.729, Val acc: 0.386
Iter 38 ,Train loss: 1.542, Train acc: 0.455, Val loss: 1.727, Val acc: 0.387
Iter 39 ,Train loss: 1.538, Train acc: 0.456, Val loss: 1.726, Val acc: 0.389
Iter 40 ,Train loss: 1.534, Train acc: 0.456, Val loss: 1.725, Val acc: 0.388
Iter 41 ,Train loss: 1.530, Train acc: 0.459, Val loss: 1.725, Val acc: 0.389
Iter 42 ,Train loss: 1.526, Train acc: 0.460, Val loss: 1.727, Val acc: 0.388
Iter 43 ,Train loss: 1.522, Train acc: 0.461, Val loss: 1.725, Val acc: 0.392
Iter 44 ,Train loss: 1.518, Train acc: 0.464, Val loss: 1.725, Val acc: 0.389
Iter 45 ,Train loss: 1.514, Train acc: 0.466, Val loss: 1.724, Val acc: 0.391
Iter 46 ,Train loss: 1.509, Train acc: 0.465, Val loss: 1.725, Val acc: 0.392
Iter 47 ,Train loss: 1.505, Train acc: 0.466, Val loss: 1.725, Val acc: 0.388
Iter 48 ,Train loss: 1.501, Train acc: 0.468, Val loss: 1.724, Val acc: 0.389
Iter 49 ,Train loss: 1.498, Train acc: 0.470, Val loss: 1.725, Val acc: 0.387
Iter 50 ,Train loss: 1.494, Train acc: 0.473, Val loss: 1.726, Val acc: 0.391
Iter 51 ,Train loss: 1.491, Train acc: 0.474, Val loss: 1.725, Val acc: 0.397
Iter 52 ,Train loss: 1.487, Train acc: 0.476, Val loss: 1.726, Val acc: 0.395
Iter 53 ,Train loss: 1.483, Train acc: 0.477, Val loss: 1.727, Val acc: 0.391
Iter 54 ,Train loss: 1.479, Train acc: 0.480, Val loss: 1.728, Val acc: 0.393
Iter 55 ,Train loss: 1.476, Train acc: 0.481, Val loss: 1.728, Val acc: 0.395
Iter 56 ,Train loss: 1.472, Train acc: 0.483, Val loss: 1.728, Val acc: 0.393
Iter 57 ,Train loss: 1.469, Train acc: 0.485, Val loss: 1.729, Val acc: 0.394
Iter 58 ,Train loss: 1.465, Train acc: 0.486, Val loss: 1.729, Val acc: 0.393
Iter 59 ,Train loss: 1.462, Train acc: 0.487, Val loss: 1.731, Val acc: 0.392
Iter 60 ,Train loss: 1.459, Train acc: 0.488, Val loss: 1.731, Val acc: 0.393
Iter 61 ,Train loss: 1.455, Train acc: 0.490, Val loss: 1.732, Val acc: 0.391
Iter 62 ,Train loss: 1.452, Train acc: 0.492, Val loss: 1.731, Val acc: 0.392
Iter 63 ,Train loss: 1.448, Train acc: 0.494, Val loss: 1.732, Val acc: 0.392
Iter 64 ,Train loss: 1.445, Train acc: 0.494, Val loss: 1.735, Val acc: 0.391
Iter 65 ,Train loss: 1.442, Train acc: 0.495, Val loss: 1.734, Val acc: 0.390
Iter 66 ,Train loss: 1.439, Train acc: 0.496, Val loss: 1.737, Val acc: 0.388
```

```
Iter 67 ,Train loss: 1.436, Train acc: 0.498, Val loss: 1.737, Val acc: 0.385
Iter 68 ,Train loss: 1.434, Train acc: 0.499, Val loss: 1.739, Val acc: 0.387
Iter 69 ,Train loss: 1.430, Train acc: 0.499, Val loss: 1.740, Val acc: 0.390
Iter 70 ,Train loss: 1.427, Train acc: 0.502, Val loss: 1.740, Val acc: 0.385
Iter 71 ,Train loss: 1.424, Train acc: 0.503, Val loss: 1.741, Val acc: 0.386
Iter 72 ,Train loss: 1.421, Train acc: 0.504, Val loss: 1.742, Val acc: 0.389
Iter 73 ,Train loss: 1.418, Train acc: 0.505, Val loss: 1.743, Val acc: 0.389
Iter 74 ,Train loss: 1.414, Train acc: 0.506, Val loss: 1.743, Val acc: 0.389
Iter 75 ,Train loss: 1.412, Train acc: 0.507, Val loss: 1.746, Val acc: 0.390
Iter 76 ,Train loss: 1.409, Train acc: 0.507, Val loss: 1.748, Val acc: 0.388
Iter 77 ,Train loss: 1.405, Train acc: 0.510, Val loss: 1.749, Val acc: 0.390
Iter 78 ,Train loss: 1.404, Train acc: 0.511, Val loss: 1.751, Val acc: 0.388
Iter 79 ,Train loss: 1.400, Train acc: 0.512, Val loss: 1.753, Val acc: 0.390
Iter 80 ,Train loss: 1.397, Train acc: 0.512, Val loss: 1.752, Val acc: 0.393
Iter 81 ,Train loss: 1.394, Train acc: 0.513, Val loss: 1.753, Val acc: 0.393
Iter 82 ,Train loss: 1.392, Train acc: 0.513, Val loss: 1.753, Val acc: 0.393
Iter 83 ,Train loss: 1.389, Train acc: 0.513, Val loss: 1.752, Val acc: 0.394
Iter 84 ,Train loss: 1.386, Train acc: 0.514, Val loss: 1.755, Val acc: 0.395
Iter 85 ,Train loss: 1.383, Train acc: 0.514, Val loss: 1.755, Val acc: 0.396
Iter 86 ,Train loss: 1.380, Train acc: 0.514, Val loss: 1.755, Val acc: 0.393
Iter 87 ,Train loss: 1.378, Train acc: 0.516, Val loss: 1.754, Val acc: 0.395
Iter 88 ,Train loss: 1.375, Train acc: 0.515, Val loss: 1.754, Val acc: 0.398
Iter 89 ,Train loss: 1.372, Train acc: 0.517, Val loss: 1.752, Val acc: 0.399
Iter 90 ,Train loss: 1.369, Train acc: 0.520, Val loss: 1.756, Val acc: 0.403
Iter 91 ,Train loss: 1.367, Train acc: 0.520, Val loss: 1.755, Val acc: 0.402
Iter 92 ,Train loss: 1.365, Train acc: 0.521, Val loss: 1.760, Val acc: 0.402
Iter 93 ,Train loss: 1.362, Train acc: 0.522, Val loss: 1.761, Val acc: 0.405
Iter 94 ,Train loss: 1.360, Train acc: 0.523, Val loss: 1.764, Val acc: 0.400
Iter 95 ,Train loss: 1.357, Train acc: 0.524, Val loss: 1.764, Val acc: 0.402
Iter 96 ,Train loss: 1.355, Train acc: 0.523, Val loss: 1.766, Val acc: 0.402
Iter 97 ,Train loss: 1.352, Train acc: 0.524, Val loss: 1.763, Val acc: 0.405
Iter 98 ,Train loss: 1.350, Train acc: 0.526, Val loss: 1.765, Val acc: 0.405
Iter 99 ,Train loss: 1.347, Train acc: 0.527, Val loss: 1.768, Val acc: 0.404
```

5.1 2.3 Plotting training process

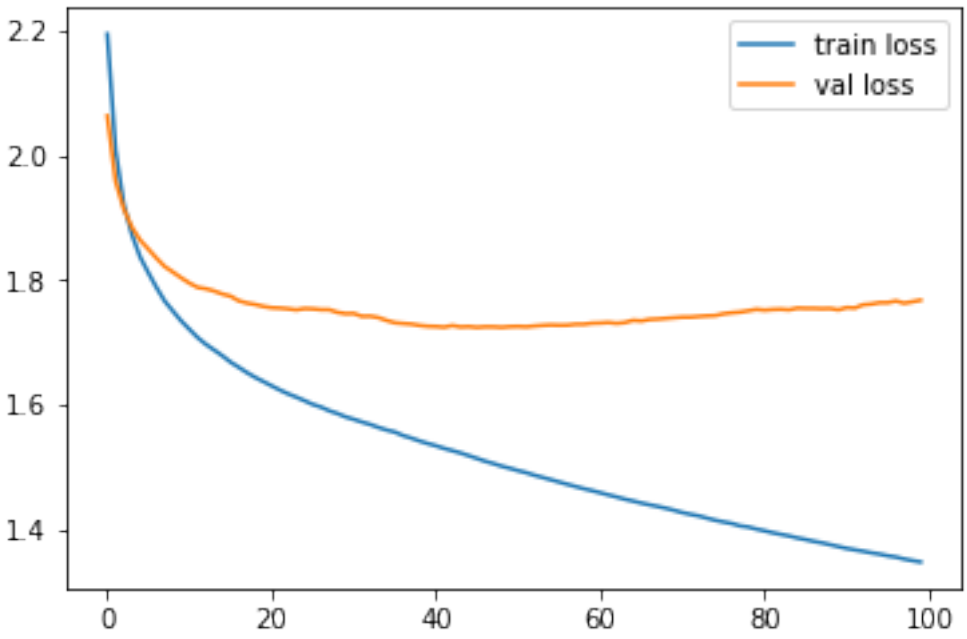
We want to first plot training loss versus validation loss, then plot training accuracy, validation accuracy.

6 TODO

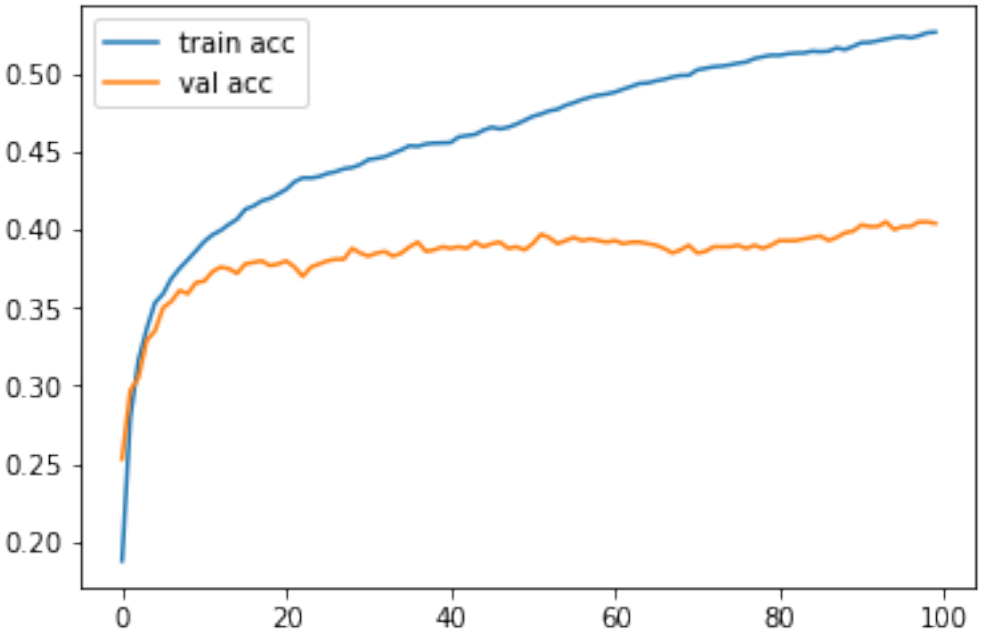
run the plotting functions

```
In [17]: ## plot training loss versus validation loss
         x_axis = np.arange(n_iteration)
         plt.plot(x_axis, train_loss_list, label='train loss')
         plt.plot(x_axis, val_loss_list, label='val loss')
```

```
plt.legend()
plt.show()
```



In [18]:
```
## plot training accuracy versus validation accuracy
plt.plot(x_axis, train_accu_list, label='train acc')
plt.plot(x_axis, val_accu_list, label='val acc')
plt.legend()
plt.show()
```



6.1 2.4 Report testing performance

We want run the trained model on test data to see how well it does

7 TODO

write code here to report test accuracy on the 10000 test data

```
In [20]: ## test performance on the test set and report final performance

         print("Test accuracy: " )
```

```
Test accuracy:  0.3903
```

7.1 3.1 Now we design a convolutional network

Check pytorch documentation for usage of each layer

8 TODO

write network structure here you should do: 1. use at least 2 conv layer 2. at least 1 maxpool layer 3. at least 2 linear layer at the end

```
In [21]: class ConvNet(nn.Module):
             def __init__(self):
                 super(ConvNet, self).__init__()


             def forward(self, x):

                 return x
```

8.1 3.2 Training the Convolutional Net model

1. We first initialize the neural network, and also the optimizer and the criterion (loss function)
2. We then convert data into correct tensor form
3. We then start doing training iterations

9 TODO

write code here 1. init your model 2. use Adam optimizer, with lr=1e-3 3. use cross entropy criterion 4. for X_train, X_val, cast to Tensor 5. for y_train, y_val, cast to Tensor and then cast to type long 6. simply do 40 iterations

```
In [ ]: # init network
        conv_net = ConvNet()
        print('model structure: ', conv_net)
        # init optimizer
        optimizer =
        # set loss function
        criterion =

        # prepare for mini-batch stochastic gradient descent
        n_iteration = 40
        batch_size = 256
        n_data = X_train_normalized.shape[0]
        n_batch = int(np.ceil(n_data/batch_size))

        # convert X_train and X_val to tensor and flatten them
        X_train_tensor =
        X_val_tensor =

        # convert training label to tensor and to type long
        y_train_tensor =
        y_val_tensor =

        print('X train tensor shape:', X_train_tensor.shape)
```

10 TODO

1. Here you will have to write training code
2. As well as code to log the training process for later plotting
3. After each iteration through training data, you should use that print function to log you current training process

```
In [23]: ## start
         train_loss_list = np.zeros(n_iteration)
         train_accu_list = np.zeros(n_iteration)
         val_loss_list = np.zeros(n_iteration)
         val_accu_list = np.zeros(n_iteration)
         for i in range(n_iteration):
             # first get a minibatch of data
             for j in range(n_batch):
                 batch_start_index = j*batch_size
                 # get data batch from the normalized data
                 X_batch = X_train_tensor[batch_start_index:batch_start_index+batch_size]
                 # get ground truth label y
                 y_batch = y_train_tensor[batch_start_index:batch_start_index+batch_size]

             print("Iter %d ,Train loss: %.3f, Train acc: %.3f, Val loss: %.3f, Val acc: %.3f"
                   %(i ,ave_train_loss, train_accu, val_loss, val_accu))
             ## add to the logs so that we can use them later for plotting
             train_loss_list[i] =
             train_accu_list[i] =
             val_loss_list[i] =
             val_accu_list[i] =
```

```
Iter 0 ,Train loss: 2.185, Train acc: 0.182, Val loss: 2.058, Val acc: 0.223
Iter 1 ,Train loss: 2.004, Train acc: 0.258, Val loss: 1.962, Val acc: 0.273
Iter 2 ,Train loss: 1.904, Train acc: 0.298, Val loss: 1.840, Val acc: 0.308
Iter 3 ,Train loss: 1.798, Train acc: 0.338, Val loss: 1.766, Val acc: 0.342
Iter 4 ,Train loss: 1.707, Train acc: 0.376, Val loss: 1.724, Val acc: 0.358
Iter 5 ,Train loss: 1.628, Train acc: 0.410, Val loss: 1.684, Val acc: 0.383
Iter 6 ,Train loss: 1.562, Train acc: 0.436, Val loss: 1.625, Val acc: 0.401
Iter 7 ,Train loss: 1.515, Train acc: 0.450, Val loss: 1.591, Val acc: 0.419
Iter 8 ,Train loss: 1.469, Train acc: 0.471, Val loss: 1.567, Val acc: 0.427
Iter 9 ,Train loss: 1.427, Train acc: 0.482, Val loss: 1.565, Val acc: 0.436
Iter 10 ,Train loss: 1.399, Train acc: 0.493, Val loss: 1.563, Val acc: 0.436
Iter 11 ,Train loss: 1.377, Train acc: 0.504, Val loss: 1.539, Val acc: 0.445
Iter 12 ,Train loss: 1.343, Train acc: 0.515, Val loss: 1.513, Val acc: 0.452
Iter 13 ,Train loss: 1.308, Train acc: 0.530, Val loss: 1.488, Val acc: 0.473
Iter 14 ,Train loss: 1.265, Train acc: 0.546, Val loss: 1.486, Val acc: 0.470
Iter 15 ,Train loss: 1.236, Train acc: 0.553, Val loss: 1.476, Val acc: 0.489
Iter 16 ,Train loss: 1.208, Train acc: 0.565, Val loss: 1.506, Val acc: 0.485
Iter 17 ,Train loss: 1.188, Train acc: 0.576, Val loss: 1.621, Val acc: 0.446
Iter 18 ,Train loss: 1.178, Train acc: 0.579, Val loss: 1.722, Val acc: 0.415
```

```
Iter 19 ,Train loss: 1.200, Train acc: 0.568, Val loss: 1.466, Val acc: 0.481
Iter 20 ,Train loss: 1.165, Train acc: 0.585, Val loss: 1.454, Val acc: 0.490
Iter 21 ,Train loss: 1.081, Train acc: 0.611, Val loss: 1.511, Val acc: 0.497
Iter 22 ,Train loss: 1.067, Train acc: 0.613, Val loss: 1.493, Val acc: 0.489
Iter 23 ,Train loss: 1.037, Train acc: 0.624, Val loss: 1.496, Val acc: 0.499
Iter 24 ,Train loss: 0.980, Train acc: 0.646, Val loss: 1.497, Val acc: 0.509
Iter 25 ,Train loss: 0.932, Train acc: 0.665, Val loss: 1.506, Val acc: 0.508
Iter 26 ,Train loss: 0.894, Train acc: 0.679, Val loss: 1.543, Val acc: 0.516
Iter 27 ,Train loss: 0.858, Train acc: 0.693, Val loss: 1.574, Val acc: 0.513
Iter 28 ,Train loss: 0.826, Train acc: 0.712, Val loss: 1.605, Val acc: 0.514
Iter 29 ,Train loss: 0.797, Train acc: 0.721, Val loss: 1.658, Val acc: 0.511
Iter 30 ,Train loss: 0.773, Train acc: 0.728, Val loss: 1.718, Val acc: 0.505
Iter 31 ,Train loss: 0.784, Train acc: 0.720, Val loss: 1.802, Val acc: 0.495
Iter 32 ,Train loss: 0.764, Train acc: 0.723, Val loss: 1.664, Val acc: 0.496
Iter 33 ,Train loss: 0.756, Train acc: 0.726, Val loss: 1.829, Val acc: 0.469
Iter 34 ,Train loss: 0.769, Train acc: 0.723, Val loss: 1.837, Val acc: 0.462
Iter 35 ,Train loss: 0.729, Train acc: 0.737, Val loss: 1.886, Val acc: 0.462
Iter 36 ,Train loss: 0.737, Train acc: 0.733, Val loss: 1.900, Val acc: 0.459
Iter 37 ,Train loss: 0.735, Train acc: 0.735, Val loss: 2.030, Val acc: 0.462
Iter 38 ,Train loss: 0.718, Train acc: 0.737, Val loss: 1.789, Val acc: 0.507
Iter 39 ,Train loss: 0.633, Train acc: 0.770, Val loss: 1.818, Val acc: 0.501
```

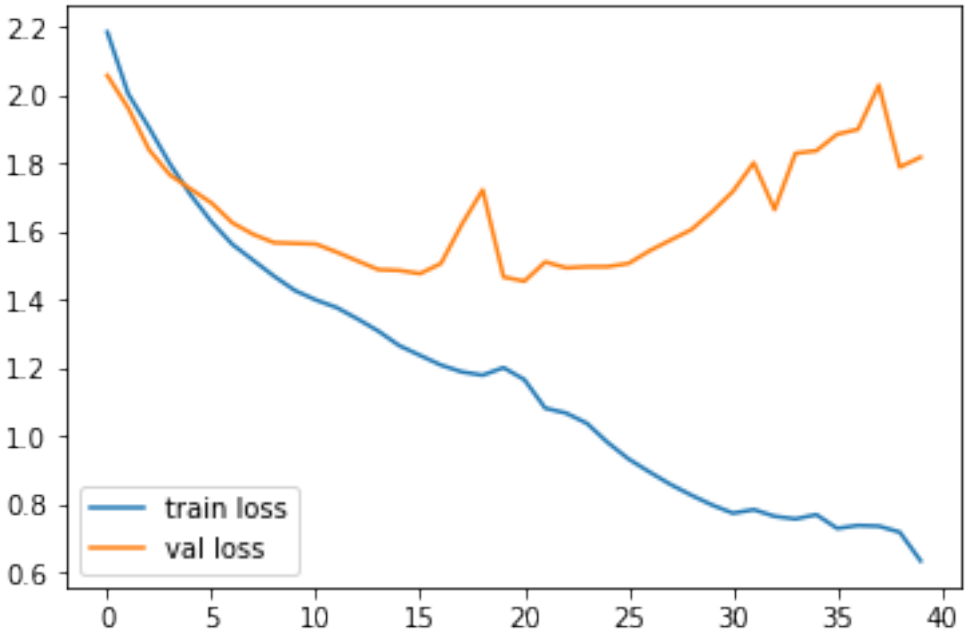
10.1 3.3 Plotting training process

We want to first plot training loss versus validation loss, then plot training accuracy, validation accuracy.

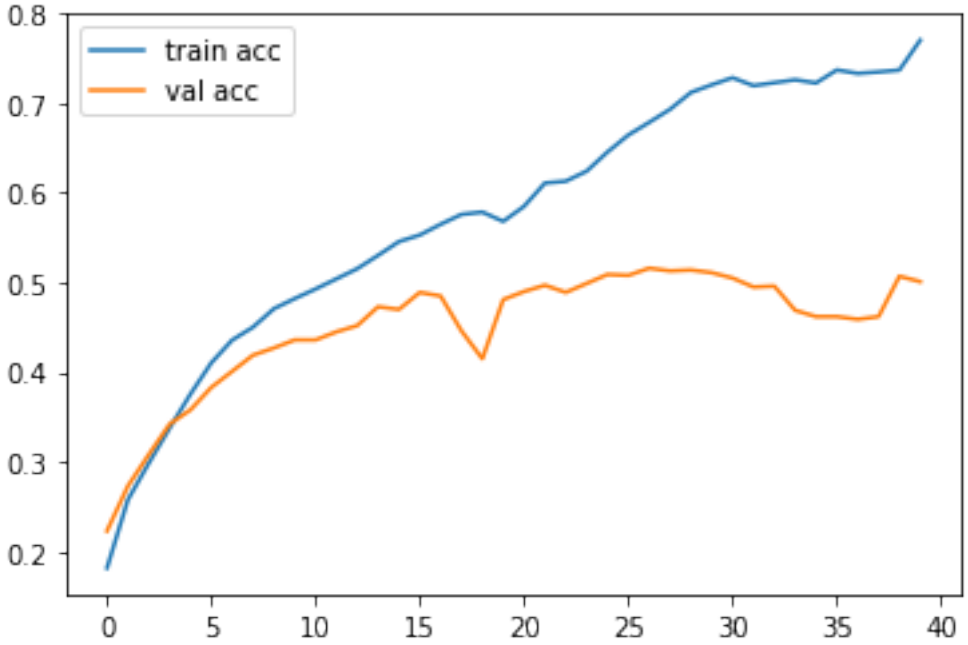
11 TODO

run the code here for plotting

```
In [24]: ## plot training loss versus validation loss
         x_axis = np.arange(n_iteration)
         plt.plot(x_axis, train_loss_list, label='train loss')
         plt.plot(x_axis, val_loss_list, label='val loss')
         plt.legend()
         plt.show()
```



In [25]:
```
## plot training accuracy versus validation accuracy
plt.plot(x_axis, train_accu_list, label='train acc')
plt.plot(x_axis, val_accu_list, label='val acc')
plt.legend()
plt.show()
```



11.1 3.4 Report testing performance

We want run the trained model on test data to see how well it does

12 TODO

write code here to report test accuracy on the 10000 test data

```
In [26]: ## test performance on the test set and report final performance
         print("Test accuracy: ", )
```

```
Test accuracy:  0.5085
```

12.1 Other things you can do to improve model performance:

1. Add in regularization, you can use l2, Dropout
2. Use a better data normalization method
3. Train on more data
4. Do data augmentation to get more data
5. Use a more powerful network structure
6. Do transfer learning