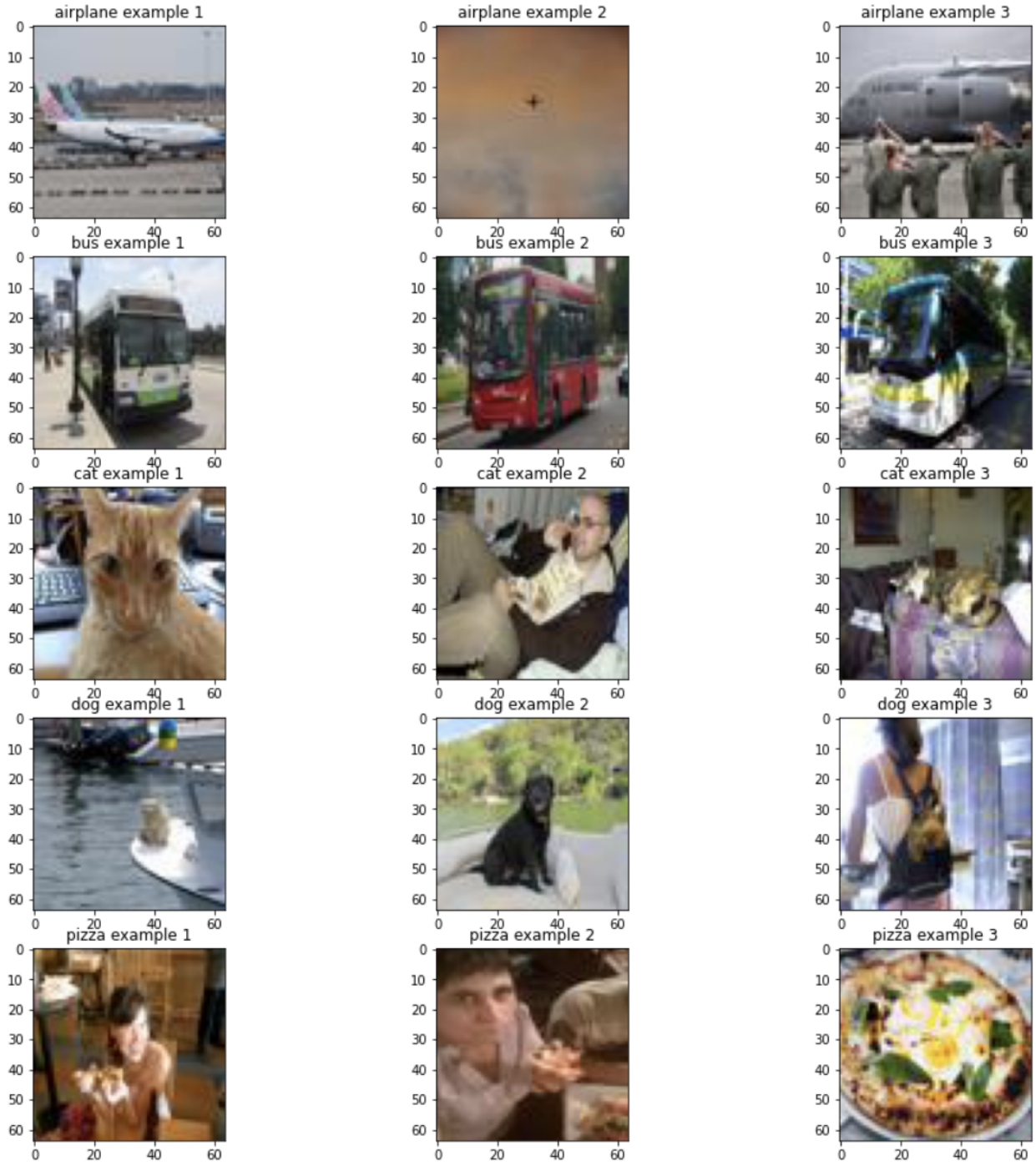
February 20, 2023

Part 1: Creating an Image Classification Dataset

The COCO API was used to download 10,000 images equally from five classes in the COCO dataset. Each of the 2,000 images from each class were split into 1,500 training images and 500 validation images. The images were re-sized to 64, 64. Below is a printout of 3 images from each of the 5 classes.

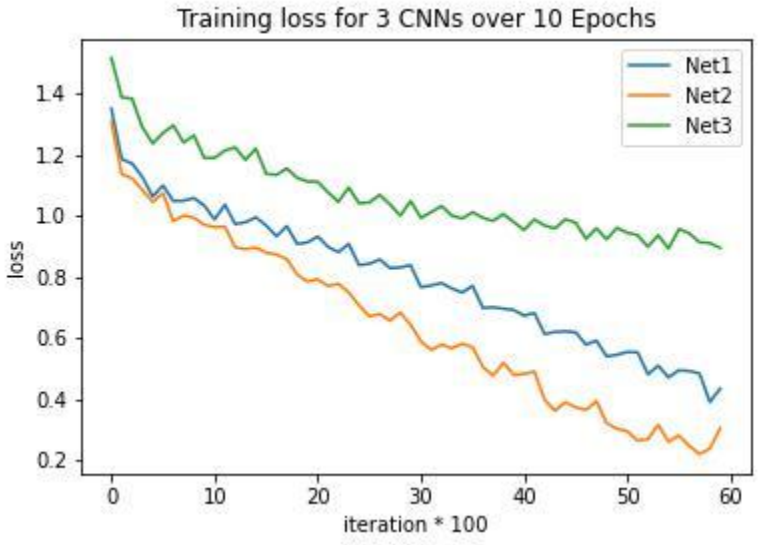


February 20, 2023

Part 2: Training 3 different CNNs and visualizing results

Below is the loss curve for each of the 3 CNNs.

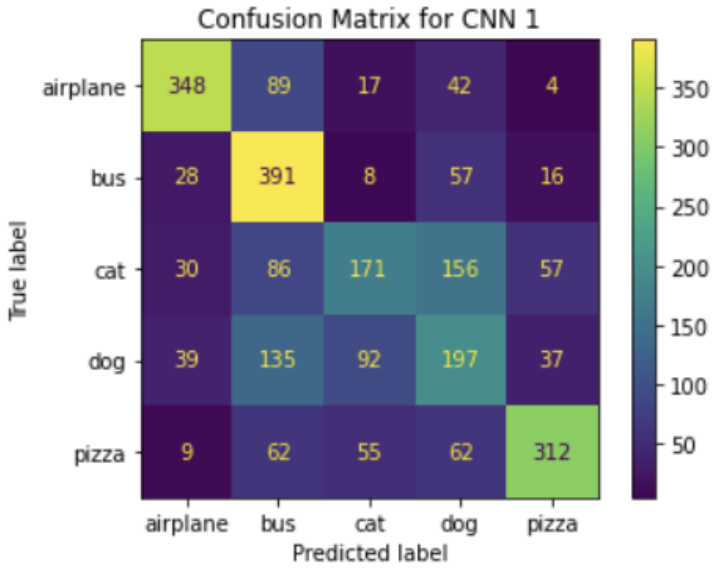
Fig 1:



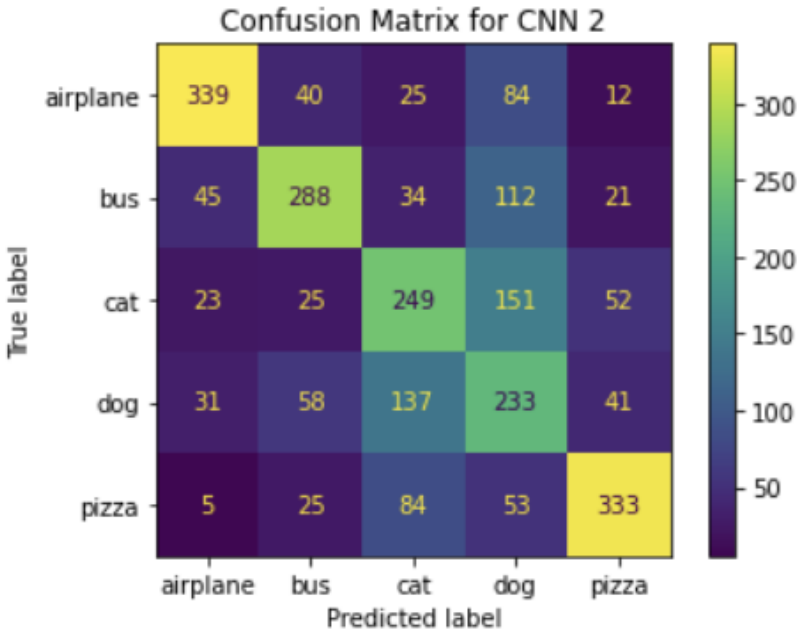
Below are the confusion matrices of the trained models' evaluations on validation sets.

February 20, 2023

```
Accuracy of the network on the val images: 56 %
```

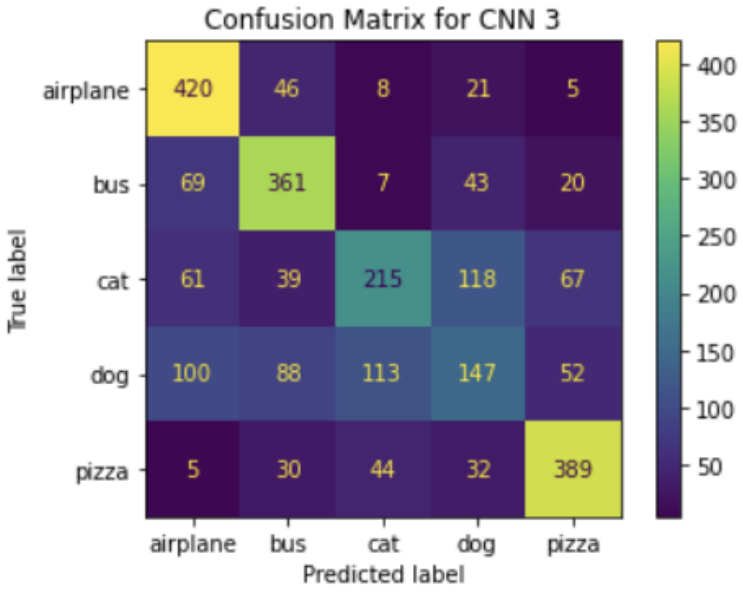



```
Accuracy of the network on the val images: 57 %
```



February 20, 2023

```
Accuracy of the network on the val images: 61 %
```



Part 3: Discussion Questions:

Does padding the convolutional layers make a difference in the classification performance?

Yes, padding did make a small difference. It improved the validation accuracy in CNN2 compared to CNN1 by 1%. Furthermore, padding was included in the deeper network (CNN3) which performed better than CNN1 and CNN2 by 5% and 4%, respectively. This is probably because padding allows the convolution output to remain the same size, which in turn might keep the important feature in the dataset that is helping with the classification decision. If we are not padding, it effectively removes some dimensionality, which could allow the network to lose some important information.

Do we observe a vanishing gradient in Net3?

Yes - it does look like Net3 had a vanishing gradient. The rate of loss during training for Net3 seemed to equal the rate of loss during training for Net 1 and Net2 for up to 5 epochs, but, after that the slope of loss decreased in Net3 compared to the Net1 and Net2 losses. The interesting fact however, was that the classification accuracy of Net3 was just about the same as Net1 and Net2, even though the loss curve was higher.

Compare the classification results of all three networks. Which CNN was the best performer?

Overall, the results of all three networks were quite similar, with only a 5% difference between Net1, Net2, and Net3. The overall accuracy (defined as total correct predictions divided by total number of items in the validation dataset) is summarized below:

Network	Total Accuracy
Net1	56%
Net2	57%
Net3	61%

If we rank the “best performer” by total accuracy, then, Net3 was the best performer.

By observing confusion matrices, which classes are more difficult to correctly differentiate and why?
In all three networks, there was difficulty in classifying dogs and cats. The best performing network on dogs was Net2, which only classified dogs correctly 46% of the time. The best performing network on cats was also Net2, which classified cats correctly 50% of the time. On the other hand, airplanes and pizza were much easier to classify, as the range of classification accuracy for airplanes was between 67% in Net1 and 84% in Net3. The range of classification accuracy for pizza was between 62% in Net1 and 78% in Net3. If the criteria for best performance is based on total overall accuracy, then, Net3 slightly beats out Net1 and Net2. However, if one were to base the criteria on accuracy of certain objects, then, below is a table of best networks for each object:

Class	Best Network	Best Networks’ Acc
Airplane	Net3	84%
Bus	Net1	78%
Cat	Net2	46%
Dog	Net2	50%
Pizza	Net3	78%

The reason cats and dogs are difficult to correctly differentiate is likely because dogs and cats look so similar to each other. If convolution is aiming to pick out differentiating features ((like eyes, legs, or ears) that minimize the loss, then it is possible that dogs and cats have a lot of similarities even after running the convolution and finding the best parameters for those convolutions. This is shown in the confusion matrix as the largest mis-classification of a cat is a dog and the largest mis-classification of a dog is a cat. Airplanes, buses, and pizza are much more distinct from each other. For example, only airplanes have wings, pizza is shaped like a triangle, and buses do not have either of those wings / triangle shapes. These distinct characteristics of airplanes, buses, and pizza are likely why the classifier had an easier time classifying those classes.

What is one thing to make classification performance better?
One thing to possibly explore is to add more training data to the network. COCO has many more images, so it would be interesting to know if training on 5,000 images per class instead of 1,500 per class would make a difference. Furthermore, adding some data augmentation transforms using tvt.Compose() could help the network learn some color and rotation invariant properties of the classification categories, which in turn might improve the classification performance better.

Part 4: Source Code

The first part of the project was to use the COCO API to download 1,500 training images and 500 validation images per class and resize the images to 64 x 64. The code below does this.

```
catNms=['airplane','bus','cat', 'dog', 'pizza']
catIds = coco.getCatIds(catNms=catNms); #get a list category ids for each category name
###Create training data set ###
for n, cat_id in enumerate(catIds): ##for each category id
    print('in category id:,',cat_id)
    imgIds = coco.getImgIds(catIds = cat_id) #for each category id, get a list of img ids
    print(imgIds[:10]) #printing first 10 image ids
    pathlib.Path('train_orig/' + catNms[n]).mkdir(parents=True, exist_ok=True) #create a path to store
#     training data for the current category
    coco.download(tarDir = 'train_orig/' + catNms[n], imgIds = imgIds[0:1500]) #download first 1500 image ids
#     into the specified directory
    d = os.listdir('train_orig/' + catNms[n]) #create list of files in the created directory. This list has 1500
#     jpg file names
#     print('list of files in d', d)
    for img in d: #iterate through list of downloaded images. Resize them to 64 x 64.
        temp_img = Image.open('train/' + catNms[n] + '/' + img) #open image
        temp_img = temp_img.resize((64,64)) #resize
        temp_img.save(fp = 'train/' + catNms[n] + '/' + img) #overwrite image with the 64 x 64 version
        ## save function parameters:
#         fp – A filename (string), pathlib.Path object or file object.
#         format – Optional format override. If omitted, the format to use is determined from the
#         ##filename extension.
#         If a file object was used instead of a filename, this parameter should always be used.

```

```
###Create validation dataset ###
catNms=['airplane','bus','cat', 'dog', 'pizza']
catIds = coco.getCatIds(catNms=catNms); #get a list category ids for each category name
for n, cat_id in enumerate(catIds): ##for each category id
    print('in category id:,',cat_id)
    imgIds = coco.getImgIds(catIds = cat_id) #for each category id, get a list of img ids
    print(imgIds[:10]) #printing first 10 image ids
    pathlib.Path('val_orig/' + catNms[n]).mkdir(parents=True, exist_ok=True) #create a path to store training data
    coco.download(tarDir = 'val_orig/' + catNms[n], imgIds = imgIds[1500:2000]) #download first 1500 image ids
#     into the specified directory
    d = os.listdir('val_orig/' + catNms[n]) #create list of files in the created directory. This list has 1500
#     jpg file names
#     print('list of files in d', d)
    for img in d: #iterate through list of downloaded images. Resize them to 64 x 64.
        temp_img = Image.open('val/' + catNms[n] + '/' + img) #open image
        temp_img = temp_img.resize((64,64)) #resize
        temp_img.save(fp = 'val/' + catNms[n] + '/' + img) #overwrite image with the 64 x 64 version
        ### save function parameters:
        #fp – A filename (string), pathlib.Path object or file object.
        #format – Optional format override. If omitted, the format to use is determined from the
        ###filename extension.
        #If a file object was used instead of a filename, this parameter should always be used.

```

Below is the code to plot 3 images of 5 different categories:

```
### Plot images from each category ###
catNms=['airplane','bus','cat', 'dog', 'pizza']
fig, ax = plt.subplots(5,3, figsize=(16, 16))
for n in range(0,5):
    temp_dir = os.listdir('train/' + catNms[n])
    #open image
    for i in range(6,9):
        temp_img = Image.open('train/' + catNms[n] + '/' + temp_dir[i]) #get the i-th image from the directory.
#         Here, we just grab the 0th, 1st, and 2nd.
        #convert to numpy array for plotting
        temp_np_arr = np.array(temp_img)
        ax[n,i-6].imshow(temp_np_arr)
        ax[n,i-6].set_title(catNms[n] + ' example ' + str(i-5) )

plt.savefig('example_training_images.jpg')

```

Below is the dataset class. This has a constructor that organizes classification categories into indices. It also creates a list of jpg files for each classification category that can be accessed via an index later on during the __getitem__ method. It also overrides the __len__ method so that the data loader knows how many iterations it should run to run through the dataset.

```
### Create data_loader ###
root_train = 'train/'
root_val = 'val/'
catNms=['airplane','bus','cat', 'dog', 'pizza']

class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root, catNms):
        super(MyDataset).__init__()
        self.root = {} #dictionary for main directory which holds all the images of a category
        self.filenames = {} #dictionary for filenames of a given category
        for cat in catNms:
            self.root[cat] = root + cat + '/'
        for cat in catNms:
        #create list of image files in each category that can be opened by __getitem__
            self.filenames[cat] = os.listdir(self.root[cat])

        self.rand_max = len(os.listdir(self.root[catNms[0]])) - 1 #number of files in directory

        self.mapping = {0 : 'airplane',
                        1: 'bus',
                        2: 'cat',
                        3: 'dog',
                        4: 'pizza'} #makes it easy to convert between index and name of a category.

        self.one_hot_encoding = {0: torch.tensor(np.array([1, 0, 0, 0, 0])),
                                 1: torch.tensor(np.array([0, 1, 0, 0, 0])),
                                 2: torch.tensor(np.array([0, 0, 1, 0, 0])),
                                 3: torch.tensor(np.array([0, 0, 0, 1, 0])),
                                 4: torch.tensor(np.array([0, 0, 0, 0, 1]))} #one hot encode each category.


        self.to_Tensor_and_Norm = tvt.Compose([tvt.ToTensor(),tvt.Resize((64,64)) ,
                                               tvt.Normalize([0], [1]) ]) #normalize and resize in case the resize op
#         wasn't done. Note that resizing here may not have any impact as the resizing was done previously.
    def __len__(self):
        count = 0
        for cat in catNms:
            temp_num = os.listdir(self.root[cat])
            count = count + len(temp_num)
        return count #return count. Will be 2500 if the root=val/ and 7500 if root=train/
```

Dataset class continued:

```
    def __getitem__(self, index):
        file_index = index % self.rand_max + 1
        class_index = index % 5

        img_file = self.filenames[self.mapping[class_index]]

        try:
            item = Image.open(self.root[self.mapping[class_index]] + img_file[file_index])
#         except IndexError:
#             print('these are the indices for the line above when shape is correct', class_index , file_index)

        np_img = np.array(item)
        shape = np_img.shape
        while shape != (64, 64 ,3): #handle if the image from COCO is grayscale.
            #print('found a grayscale image, fetching an RGB!')
            another_rand = random.randint(0,self.rand_max)  #generate another rand num
            #print('another_rand is', another_rand)
            try:
                item = Image.open(self.root[self.mapping[class_index]] + img_file[another_rand])
#             except IndexError:
#                 print('these are the indices for the line above when shape is incorrect', another_rand , class_in
            np_img = np.array(item)
            shape = np_img.shape

        img = self.to_Tensor_and_Norm(item)
        class_label = self.one_hot_encoding[class_index].type(torch.FloatTensor) #convert to Float
        return img, class_label

```

Now that the Dataset class has been created, a training_dataloader and validation_dataloader has been wrapped in the torch.utils.data.DataLoader class. I used 12 for my batch size with 4 workers.

```
# Use MyDataset class in PyTorches DataLoader functionality
my_train_dataloader = torch.utils.data.DataLoader(my_train_dataset, batch_size=12, num_workers = 4, drop_last=False
my_val_dataloader = torch.utils.data.DataLoader(my_val_dataset, batch_size = 12, num_workers = 4, drop_last = False
for n, batch in enumerate(my_train_dataloader):
#     #Note: each batch is a list of length 2. The first is a pytorch tensor B x C x H x W and the
#     #second is a pytorch tensor of length B with the associated class labels of each image in the
#     #first item of the list!
    print('batch is', n)
```

We now have enough infrastructure to load data and run it through a network. Here is the definition of Net1 (taken from the HW PDF)

```
###Create CNN ####
### Use valid mode - i.e, no padding

class HW4Net(nn.Module):
    def __init__(self):
        super(HW4Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.fc1 = nn.Linear(6272,64)
        self.fc2 = nn.Linear(64, 5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

February 20, 2023

Here is the code to run that network's forward function, compute loss, and compute gradient of loss with respect to the parameters. Furthermore, the code below runs backpropagation aims and to find the best set of parameters for 10 epochs. Note that for Net2 and Net3, the code is the same except the instance of the network class is different, so they have not been included in this report. However, all the code is in the attached .py file.

```
net1 = HW4Net()
loss_running_list_net1 = []
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net1.parameters(), lr = 1e-3, betas = (0.9, 0.99))
epochs = 10
for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(my_train_dataloader):
        inputs, labels = data
        optimizer.zero_grad() #Sets gradients of all model parameters to zero. We want to compute fresh gradients
        #based on the new forward run.
        outputs = net1(inputs)
        loss = criterion(outputs, labels) #compute cross-entropy loss
        loss.backward() #compute derivative of loss wrt each gradient.
        optimizer.step() #takes a step on hyperplane based on derivatives
        running_loss += loss.item()
        if (i+1) % 100 == 0:
            print("[epoch: %d, batch: %5d] loss: %3f" % (epoch + 1, i + 1, running_loss / 100))
            loss_running_list_net1.append(running_loss/100)
            running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for n, data in enumerate(my_val_dataloader):
            images, labels = data
            outputs = net1(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0) #add to total's total
            for n, i in enumerate(labels):
                temp = np.array(i) #temp holds the one hot encoded label
                idx = np.argmax(temp) #get the argmax of the encoded label - will be a value between 0 and 4.
                #print(idx)
                if idx == predicted[n]: #if the predicted value and label match
                    correct = correct + 1 #add to correct total

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

```

In order to compute validation accuracy at the end of the training, the following code runs the model through the validation dataset using the val_dataloader. A confusion matrix is also displayed using this code.

```python
### Test performance of CNN 1 on val data ###
correct = 0
total = 0
y_pred = []
y_label = []
mapping = { 0: 'airplane',
            1: 'bus',
            2: 'cat',
            3: 'dog',
            4: 'pizza'}


with torch.no_grad():
    for n, data in enumerate(my_val_dataloader):
        images, labels = data

        outputs = net1(images)

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

        total += labels.size(0) #add to total count of ground truth images so we can calculate total accuracy
        #print("total images in val set", total)
        for n, i in enumerate(labels):
            temp = np.array(i) #arrays are one hot encoded, we need to convert it into a human readable label for
            #display in the confusion matrix
            label_arg = np.argmax(temp) #get the argument of the one hot encoding
            y_label.append(mapping[label_arg]) #apply the argument to the mapping dictionary above. For example
            # if the argument is 3, then, that corresponds to a label of dog in the mapping dictionary
            t = int(np.array(predicted[n])) #get integer representation of prediction from network (will
            #be an int from 0 to 4.
            y_pred.append(mapping[t]) #append the predicted output of this label to the prediction list, but,
            #via the mapping dictionary definition so that the y_pred list is human readable.

            if label_arg == predicted[n]:
                correct = correct + 1 #add to total count of correct predictions so we can calculate total accuracy


print('Accuracy of the network on the val images: %d %%' % (
    100 * correct / total))
from sklearn.metrics import confusion_matrix

y_true = y_label
y_pred = y_pred
confusion_matrix=confusion_matrix(y_true, y_pred, labels = [ "airplane", "bus", "cat", "dog", "pizza"])
disp = ConfusionMatrixDisplay(confusion_matrix, display_labels = [ "airplane", "bus", "cat", "dog", "pizza"])
disp.plot()
disp.ax_.set_title("Confusion Matrix for CNN 1")
plt.show()
plt.savefig('CM_CNN1')

```

Finally, the loss curves are plotted for all three networks using the code below:

```python
### Plot training loss for CNN1 and CNN2 ###

plt.plot(loss_running_list_net1, label = 'Net1')
plt.plot(loss_running_list_net2, label = 'Net2')
plt.plot(loss_running_list_net3, label = 'Net3')
plt.xlabel('iteration * 100')
plt.ylabel('loss')
plt.title('Training loss for 3 CNNs over 10 Epochs')
plt.legend()
plt.savefig('Training_loss.jpg')

```