Chris Ardohain
BME64600/ECE60146

Homework 4: Basic CNN Construction - PyTorch

Objective:

The objective of this homework was to introduce or refamiliarize us with basic CNN construction, specifically in PyTorch using the COCO dataset as training and validation data. The project exercises our ability to read in a dataset, construct a CNN, build a training and validation wrapper, and evaluate the results.

Tasks:

Task 1: Prepare the dataset

The first step in this exercise was the preparation of the dataset for ingestion by a CNN. The homework refers to this as dataset creation, however, dataset creation typically refers to the painstaking process of label assignment to each image. In this particular case, we are using the COCO dataset which comes prelabeled. Fortunately, the COCO dataset also comes with a python specific library that greatly helps in terms of parsing the data. We were asked to select 8000 training images and 2000 validation images in the following classes: boat, couch, dog, cake, motorcycle. The output of my code (Appendix A – Code Snippet 1) for this section balanced the classes by randomly retrieving 2000 of each class in the training set and 400 of each in the validation set, however, I belatedly realized that each image had multiple images so the result was 8000 and 2000 annotations (respective to training and validation sets) but only 7862 and 1992 images respectively. It also resized the images according to the requirements set out in the homework. Lastly, it produced to csv log files, one for training and one for validation, that logged image names, assigned class, and class number. The first 3 images of each class are presented in Figure 1.

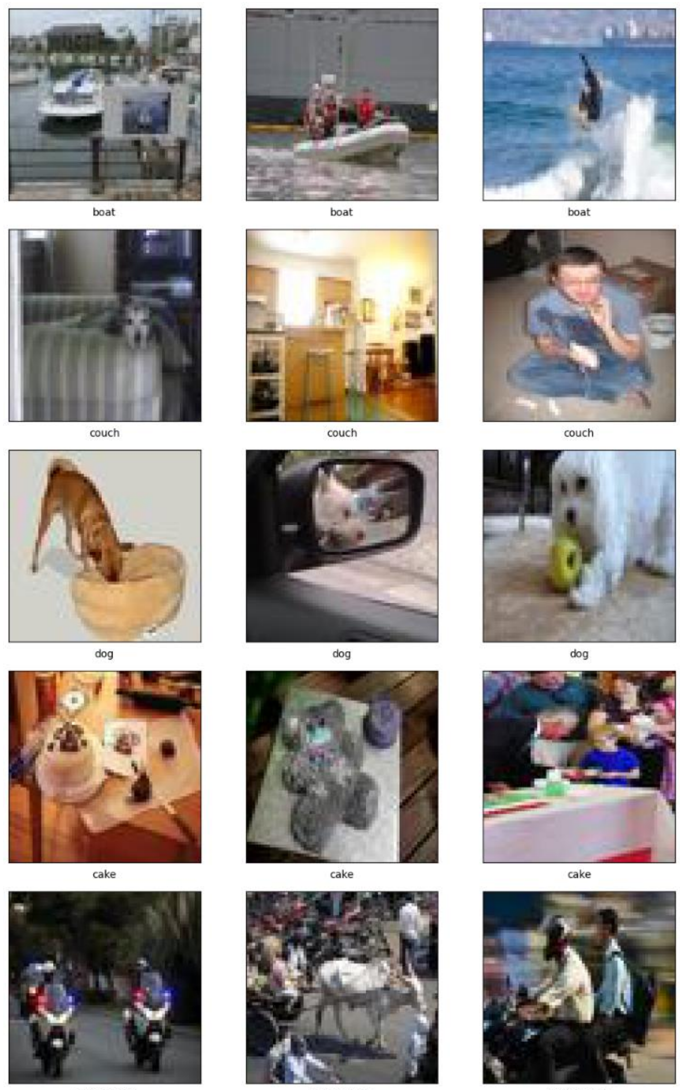

Figure 1: Examples of COCO images by required classes

Task 2: CNN Construction

CNN construction was relatively simple given the example provided in the homework. Establishing the network architecture is very much streamlined compared to the last homework through the use of Pytorch, however, it is vital to understand how information is being passed between layers and how each layer interacts and changes the data before it is passed to the next layer. To calculate the layer output size and total number of layer parameters, I used the following two equations modified with variables for my own understanding taken from here and here.

Calculate layer output size

$$\frac{W - F + 2P}{S} + 1$$

Where W is input size, F is filter size (user input), P is padding, and S is stride

Calculate layer parameters

$$OC * (KS^2 * IC + 1)$$

Where KS is kernel size, IC is input channels, and OC is output channels

Figure 2 below highlights the application of these equations to the three networks we were required to build, specifically as they relate to the homework requirements. The output feature map of the first fully connected layer were of the following sizes [6272, 8192, 6272] with respect to Net1, Net2, and Net3. For Net3, we were asked to ensure that the added convolutional layers had padding, however, we were not asked to add padding to the first two convolutional layers as compared to Net1. The total number of parameters were [406885, 535178, 499365] respectively. The final build of these networks in code form can be found in Appendix A – Code Snippet 2.

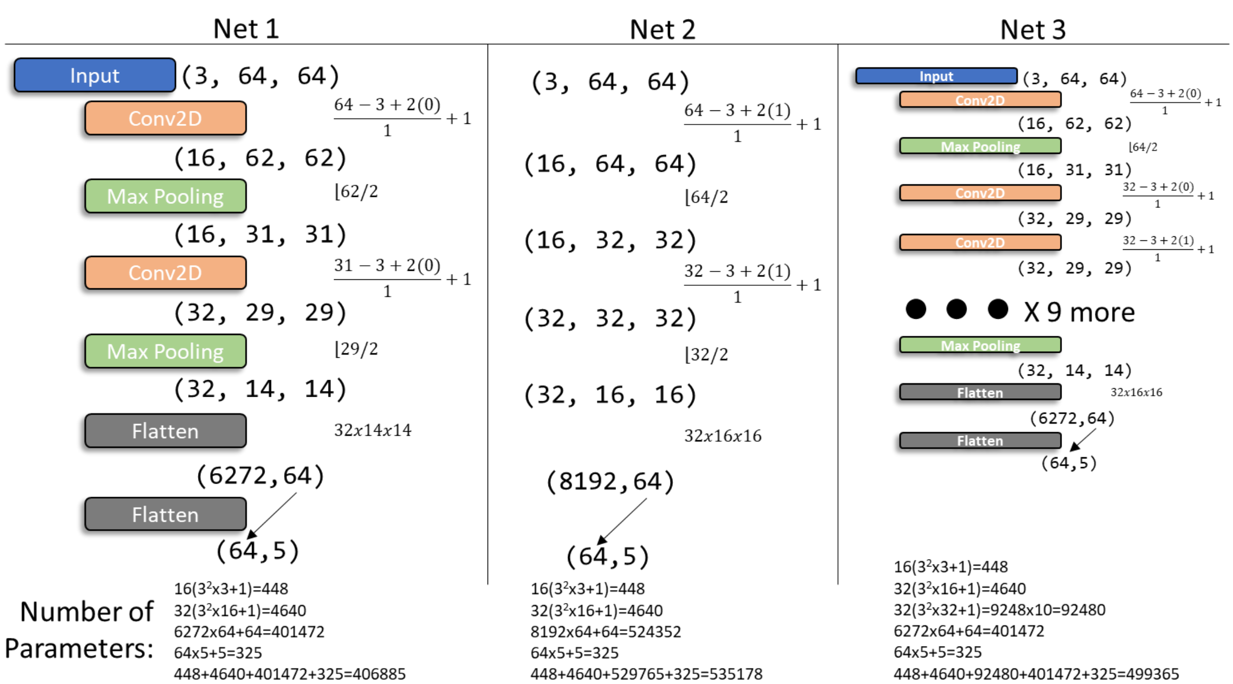

Figure 2: Calculation of layer inputs, outputs, and parameters for each required network

Task 3: Data Loading, Model Training, and Validation Inference

After the networks were built, the next step involved developing a modified data loader subclass that modifies the imagery in preparation for passing to the networks for training or evaluation. This subclass reads in the csv data saved previously when the training and validation data was appropriated and separated into new folders as well as modifying images as they are read in. There is a transform that is applied converting the image to a tensor while normalizing between the values of -1 to 1. I also noticed that there were some grayscale images (1xchannel) that needed to be converted to RGB when appropriate. This data loader subclass then passes the modified images to the DataLoader class during training. Code for the subclass can be found in Appendix A – Code Snippet 3.

The other section of this code necessitated the development of a model training function that accomplished several functions. This is the section that caused me the most trouble, so it is heavily commented, but the first step was importing the data by first assigning the data loader subclass to the correct location and then passing to the data loader base class. Next, I assigned the GPU to conduct the work before loading the network, assigning the optimizer, conducting model training, and logging the training loss. Upon completion of training, this function also switches to evaluation mode and inferences all the validation data, producing a dataframe with predictions and "ground truth" labels. I would normally split these two steps so that the model weights were saved to memory and later loaded in for validation (really testing) and application, however, that sort of workflow is unnecessary for the homework requirements so this condensed version works better. The output of this function is the loss log for every epoch and the prediction/reference log. The function itself can be found in Appendix A – Code Snippet 4.

Task 4: Model Evaluation

The last coding requirement was the evaluation of the model through traditional means. The functions I used to plot the results and the __main__ call function tying all the previous code together in one bundle can be found in Appendix A – Code Snippet 5 and 6 respectively. The plotting function produces both a loss comparison between networks, as well as confusion matrices for each network. Lastly, it produces one csv per network that provides overall accuracy per class. All tables and figures are provided below.

	COCO Class					
Network	Boat	Couch	Dog	Cake	Motorcycle	Overall
Net 1	0.613	0.540	0.343	0.558	0.555	0.522
Net 2	0.650	0.480	0.265	0.485	0.633	0.525
Net 3	0.600	0.498	0.335	0.508	0.568	0.515

Table 1: Model accuracies by COCO class and CNN

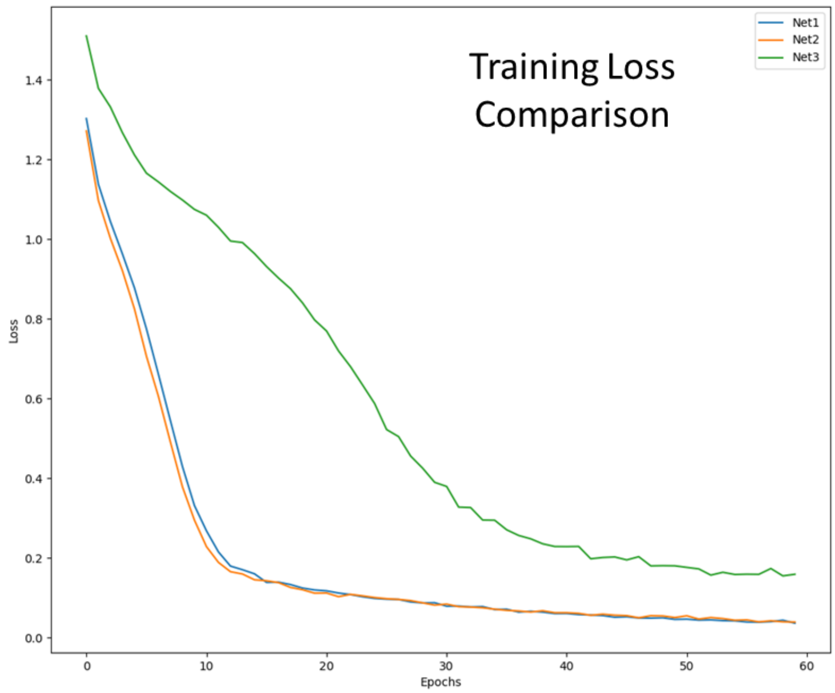

Figure 3: Training Loss Comparison between the three networks.

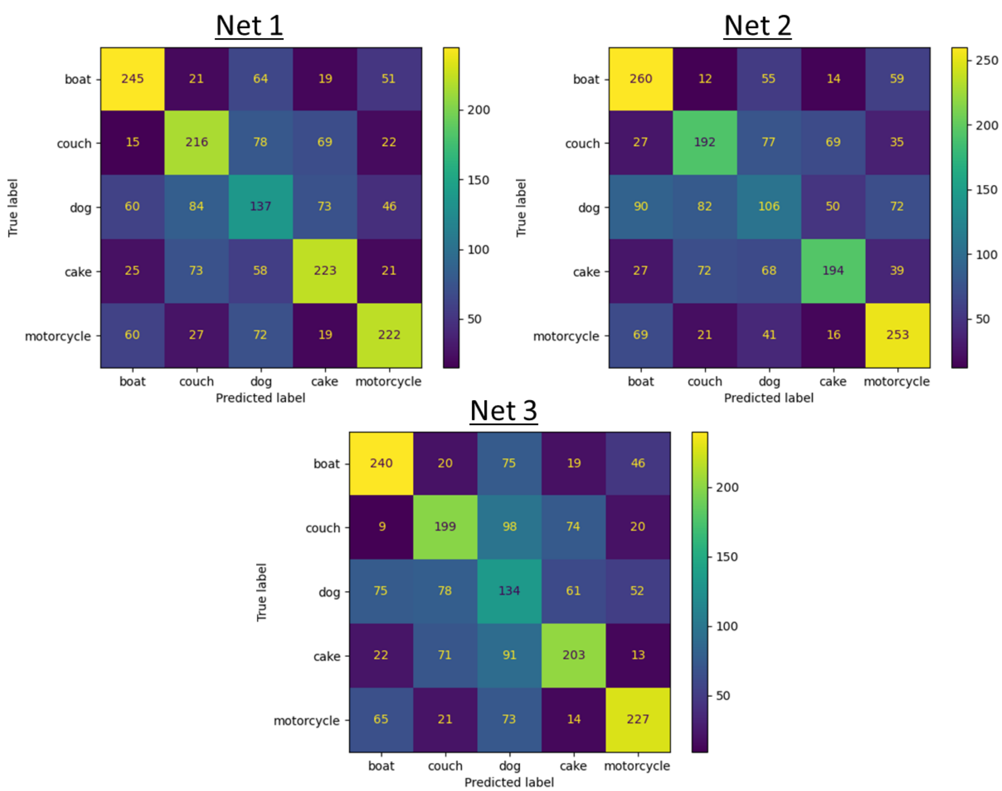

Figure 4: Confusion matrix for each model.

Chris Ardohain
BME64600/ECE60146

Discussion Questions:

1. My results suggest that adding padding didn't make much of a difference in terms of classification performance, however, I understand that this is not always the case. Padding ensures minimal loss of data when passing through convolutional layers which may be important depending on your application. My intuition is that padding becomes more important either when image resolution is poor or when object relationships within the images are complex. In the first case, each pixel gains in importance since there are so few to spare, while in the second case, loss of data may blur the relationship between objects.
2. Net 3 does have a vanishing gradient problem as highlighted by the resultant loss curve which is not a smooth downward slope toward convergence. I was actually surprised that Net 3 didn't result in over training, although it is likely if the epoch count was increased beyond the 60 I ran for my models. Net 3 just has too many layers for this application, and it highlights that when it comes to CNNs, increased complexity isn't always better.
3. Results indicate similar performance between all three networks. Network 1 is the best performing, followed by network 2 and then network 3, but the differences in accuracy are negligible. Network 1 performed best in three of the five classes (couch, dog, cake) while network 2 performed best in the other two of five classes (boat, motorcycle) but worst in the other three.
4. The easiest class to identify across all three networks was boat which I would expect would relate to image context since most boat pictures are going to include water which is a rather discernable feature as compared to the environments in which the other classes could be found. For a similar reason, motorcycles are next which are always going to be found in areas with roads, asphalt, or concrete. Conversely, couches, dogs, and cakes are found in environments that are highly varied (wall color, rug color, furniture, people, etc.).
5. For application, I would suggest moving forward with network 1 as it is the simplest of the networks that also happens to produce the best results if we continued to work with 64x64 images. To improve performance, I would first add in all training data that contained labels of the classes of interest from the COCO dataset. Assuming we are continuing to work with 64x64 images, I would next focus on hyperparameter tuning. Lastly, I would apply cross validation to ensure the model had an opportunity to train on all available images given the limited number available. This would require the separation of a true test set but might be worth it in the end.

Appendix A – Code

```python
#Import Statements
import os
from PIL import Image
import matplotlib.pyplot as plt
import pandas as pd
from pycocotools.coco import COCO
import random
import torchvision.transforms as tvt
import torch
########################
#Set initial variables
annotations_path=r"C:/BME_646/coco_dataset/annotations_trainval2017/annotations/instances_train2017.json"
images_path=r"C:/BME_646/coco_dataset/train2017/"
hw_train_path=r"C:/BME_646/coco_dataset/hw_subset/train/"
hw_val_path=r"C:/BME_646/coco_dataset/hw_subset/val/"
csv_out_path=r"C:/BME_646/coco_dataset/hw_subset/"
categories=['boat','couch','dog','cake','motorcycle']
num_train=8000
num_val=2000
num_figs=15
########################
#Load the Annotation file using COCO
annotations=COCO(annotations_path)
########################
#Load images, resize and save them
i=0
train_temp=[]
val_temp=[]
for category in categories:
    #reassign class numbers
    cls_num=i
    i+=1
    #Create a list of image names based on category
    img_ids=annotations.getImgIds(catIds=annotations.getCatIds(catNms=[category]))
    #creating balanced respresntation of classes in train and val datasets
    print("Working on {}s.  Be patient!".format(category))
    img_ids=random.sample(img_ids, int((num_train+num_val)/len(categories)))
    j=0
    for img_id in img_ids:
        j+=1
        #Load image
        img_meta=annotations.loadImgs(img_id)[0]
        temp_img=Image.open(images_path+img_meta['file_name'])
        #resize
        temp_img=temp_img.resize((64,64))
        if j<=int(num_train/len(categories)):
            train_temp.append({'img_num':img_meta['file_name'],'class':category,'class_num':cls_num})
            temp_img.save(hw_train_path+img_meta['file_name'])
        else:
            val_temp.append({'img_num':img_meta['file_name'],'class':category,'class_num':cls_num})
            temp_img.save(hw_val_path+img_meta['file_name'])
#Convert training and val lists to dataframes and write to csv
#to parse during data loading
train_df=pd.DataFrame(train_temp)
val_df=pd.DataFrame(val_temp)
train_df.to_csv(csv_out_path+"training_log.csv", index=False)
val_df.to_csv(csv_out_path+"validation_log.csv", index=False)
########################
#Plot the first 3 images per class
three_instances=train_df.groupby('class').head(3)
fig, axes = plt.subplots(nrows=5, ncols=3, figsize=(10,15))
axes=axes.flatten()
#Loop through the DataFrame and plot each image
for i, (fig, (_, row)) in enumerate(zip(axes, three_instances.iterrows())):
    img_path = hw_train_path+row['img_num']
    img = Image.open(img_path)
    fig.imshow(img)
    fig.set_xlabel(row['class'])
    fig.xaxis.set_label_position('bottom')
    fig.tick_params(axis='both', which='both', length=0)  # Hide tick marks
    fig.xaxis.set_ticks_position('none')
    fig.yaxis.set_ticks_position('none')
    fig.set_yticklabels([])
    fig.set_xticklabels([])
#Adjust layout
plt.tight_layout()
plt.show()
```

Code Snippet 1: Dataset preparation code - Selection and delineation of training and validation sets from the COCO dataset as well as plotting of three examples from each class.

```python
########## BUILD NETWORKS
#Establish Network 1 - From Homework
class Net1(nn.Module):
    def __init__(self):
        super(Net1,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)
    #How data is fed through the network based on attributes of class
    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

#Establish Network 2 - From Homework
class Net2(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d (16, 32, 3, padding=1)
        self.fc1 = nn.Linear(8192, 64)
        self.fc2 = nn.Linear(64,5)
    #How data is fed through the network based on attributes of class
    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
#Establish Network 3 - Modification from Net2
class Net3(nn.Module):
    def __init__(self):
        super(Net3,self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d (16, 32, 3)
        self.extra_conv1 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv2 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv3 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv4 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv5 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv6 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv7 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv8 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv9 = nn.Conv2d (32, 32, 3, padding=1)
        self.extra_conv10 = nn.Conv2d (32, 32, 3, padding=1)
        self.fc1 = nn.Linear(6272, 64)
        self.fc2 = nn.Linear(64,5)
    #How data is fed through the network based on attributes of class
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = F.relu(self.extra_conv1(x))
        x = F.relu(self.extra_conv2(x))
        x = F.relu(self.extra_conv3(x))
        x = F.relu(self.extra_conv4(x))
        x = F.relu(self.extra_conv5(x))
        x = F.relu(self.extra_conv6(x))
        x = F.relu(self.extra_conv7(x))
        x = F.relu(self.extra_conv8(x))
        x = F.relu(self.extra_conv9(x))
        x = F.relu(self.extra_conv10(x))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Code Snippet 2: Network architectures built Pytorch

```python
########## DATA LOADING
#Establish the image to tensor transformation
transform_comp=tvt.Compose([tvt.ToTensor(),tvt.Normalize(mean =[0.5], std =[0.5])])

#Establish Dataset Class
class cocoData_loader(Dataset):
    def __init__(self, csv_path, img_dir, transform=transform_comp):
        #Passed Attributes
        self.csv_data=pd.read_csv(csv_path)
        self.img_dir=img_dir
        self.transform=transform
        #Derived attributes
        self.img_num=self.csv_data['img_num']
        self.class_name=self.csv_data['class']
        self.class_num=self.csv_data['class_num']

    def __len__(self):
        return (len(self.csv_data))

    def __getitem__(self, row):
        temp_img=Image.open(self.img_dir+self.img_num[row])
        label=torch.tensor(self.class_num[row], dtype=torch.int64)
        #I came across some images that were gray, so these need
        #to be converted to RGB (e.g., 4971, 5324, etc.)
        if temp_img.mode=='L':
            temp_img=temp_img.convert('RGB')
        #Apply transform
        temp_img=self.transform(temp_img)
        return temp_img, label
```

Code Snippet 3: Data Loader subclass designed to read in csv data and modify images before passing to DataLoader base class.

```python
########## MODEL TRAINING
def train_model(csv_dir, img_dir, network, epochs, batch_s=32, learning_rate=1e-3):
#### MODEL TRAINING
    #Import the data
    train_set = cocoData_loader(csv_path = os.path.join(csv_dir,'training_log.csv'),
                                img_dir = os.path.join(img_dir,'train/'))
    train_loader = DataLoader(train_set, batch_size = batch_s, shuffle = True, num_workers = 4)

    val_set = cocoData_loader(csv_path = os.path.join(csv_dir,'validation_log.csv'),
                                img_dir = os.path.join(img_dir,'val/'))
    val_loader = DataLoader(val_set, batch_size = batch_s, shuffle = False, num_workers = 4)
    #Assign gpu if available
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    #Pass the model to the gpu
    model = network.to(device)
    criterion = nn.CrossEntropyLoss()
    #Assign Torches built in Adam optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate, betas=(0.9, 0.99))
    #Begin model training
    model.train()
    #Keep a record of the loss
    loss_log = []
    for epoch in range(epochs):
        running_loss = 0.0
        epoch_loss = 0.0
        for data in train_loader:
            #Pass the inputs
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            #Clear old gradients from last epoch
            optimizer.zero_grad()
            #conduct the forward pass
            outputs = model(inputs)
            #Calculate the batch loss
            loss = criterion(outputs, labels.long())
            #Calculate the loss with respect to the parameters
            loss.backward()
            #update the parameters accordingly
            optimizer.step()
            #record the loss for posterity
            running_loss += loss.item()
            epoch_loss += loss.item()
        #Update the loss list for later plotting
        loss_log.append(epoch_loss / len(train_loader))
        print('Epoch {} Complete - Epoch Loss= {:.2f}  ,  Total Loss={:.2f}'.format(str(epoch),epoch_loss,running_loss))
    print('Finished Training')

##### VALIDATION DATASET
    #Convert model to evaluation mode
    model.eval()
    #Initialize empty results
    total_correct = 0
    total_samples = 0
    pred_list = []
    ref_list = []

    with torch.no_grad():
        for images, labels in val_loader:
            #Pass the inputs
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            #Apply softmax and find the maximum probability class
            _, predicted = torch.max(F.softmax(outputs, dim=1), 1)
            #update the result parameters
            total_correct += (predicted == labels).sum().item()
            total_samples += labels.size(0)
            #Append predictions and references to the lists
            pred_list.extend(predicted.cpu().numpy())
            ref_list.extend(labels.cpu().numpy())
    #Convert lists to DataFrame
    pred_ref = pd.DataFrame({'pred': pred_list, 'ref': ref_list})
    accuracy = 100 * total_correct / total_samples
    print(f'Accuracy on {total_samples} validation images: {accuracy:.2f} %')

    return loss_log, pred_ref
```

Code Snippet 4: Data Loader and Validation Inference

```python
########## MODEL TRAINING
def train_model(csv_dir, img_dir, network, epochs, batch_s=32, learning_rate=1e-3):
#### MODEL TRAINING
    #Assign gpu if available
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    #Import the data
    train_set = cocoData_loader(csv_path = os.path.join(csv_dir,'training_log.csv'),
                                img_dir = os.path.join(img_dir,'train/'))
    train_loader = DataLoader(train_set, batch_size = batch_s, shuffle = True, num_workers = 4)

    val_set = cocoData_loader(csv_path = os.path.join(csv_dir,'validation_log.csv'),
                              img_dir = os.path.join(img_dir,'val/'))
    val_loader = DataLoader(val_set, batch_size = batch_s, shuffle = False, num_workers = 4)
    #Pass the model to the gpu
    model = network.to(device)
    criterion = nn.CrossEntropyLoss()
    #Assign Torches built in Adam optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate, betas=(0.9, 0.99))
    #Begin model training
    model.train()
    #Keep a record of the loss
    loss_log = []
    for epoch in range(epochs):
        running_loss = 0.0
        epoch_loss = 0.0
        for data in train_loader:
            #Pass the inputs
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            #Clear old gradients from last epoch
            optimizer.zero_grad()
            #conduct the forward pass
            outputs = model(inputs)
            #Calculate the batch loss
            loss = criterion(outputs, labels.long())
            #Calculate the loss with respect to the parameters
            loss.backward()
            #update the parameters accordingly
            optimizer.step()
            #record the loss for posterity
            running_loss += loss.item()
            epoch_loss += loss.item()
        #Update the loss list for later plotting
        loss_log.append(epoch_loss / len(train_loader))
        print('Epoch {} Complete - Epoch Loss= {:.2f}  ,  Total Loss={:.2f}'.format(str(epoch),epoch_loss,running_loss))
    print('Finished Training')

##### VALIDATION DATASET
    #Convert model to evaluation mode
    model.eval()
    #Initialize empty results
    total_correct = 0
    total_samples = 0
    pred_list = []
    ref_list = []

    with torch.no_grad():
        for images, labels in val_loader:
            #Pass the inputs
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            #Apply softmax and find the maximum probability class
            _, predicted = torch.max(F.softmax(outputs, dim=1), 1)
            #update the result parameters
            total_correct += (predicted == labels).sum().item()
            total_samples += labels.size(0)
            #Append predictions and references to the lists
            pred_list.extend(predicted.cpu().numpy())
            ref_list.extend(labels.cpu().numpy())
    #Convert lists to DataFrame
    pred_ref = pd.DataFrame({'pred': pred_list, 'ref': ref_list})
    accuracy = 100 * total_correct / total_samples
    print(f'Accuracy on {total_samples} validation images: {accuracy:.2f} %')

    return loss_log, pred_ref
```

Code Snippet 5: Model training and accuracy calculations against the validation set

```python
########## MODEL EVALUATION
def plot_results(loss_logs,preds_refs,epoch_num):
    model_names=['Net1','Net2','Net3']
    # Plotting Loss
    plt.figure(figsize=(12, 10))
    for i, loss_log in enumerate(loss_logs):
        plt.plot(range(epoch_num), loss_log, label=model_names[i])
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig('C:/BME_646/coco_dataset/output/Loss_Comparison.png')
    #plt.show()
    #Plotting Confusion Matrix
    categories = ['boat','couch','dog','cake','motorcycle']
    for preds_ref, net_name in zip(preds_refs, model_names):
        cm = confusion_matrix(preds_ref['ref'].values, preds_ref['pred'].values)
        class_acc = pd.DataFrame({'class': categories, 'accuracy': [cm[i][i] / sum(cm[i]) for i in range(len(categories))]})
        class_acc.to_csv('C:/BME_646/coco_dataset/output/class_accuracy_{}.csv'.format(net_name), index=False)
        cm_plot = ConfusionMatrixDisplay(cm, display_labels=categories).plot()
        cm_plot.figure_.savefig('C:/BME_646/coco_dataset/output/confusion_matrix_{}.png'.format(net_name))
```

Code Snippet 6: Plotting Function for both loss comparison and confusion matrices.

```python
########## RUN ALL THIS STUFF
#Do all the above when this script is called
if __name__=='__main__':
    #Initialize some hyperparameters
    epoch_num=60
    batch_s=32
    learning_rate=1e-3
    csv_log_root='C:/BME_646/coco_dataset/hw_subset/csv_logs'
    data_root='C:/BME_646/coco_dataset/hw_subset'
    # Run All three models
    results = [train_model(csv_log_root, data_root, net, epoch_num) for net in [Net1(), Net2(), Net3()]]
    # Splitting the results into separate lists
    loss_lists, preds_refs = zip(*results)
    # Saving predictions references
    for preds_ref, i in zip(preds_refs, range(1, 4)):
        preds_ref.to_csv('C:/BME_646/coco_dataset/output/pred_ref{}.csv'.format(i), index=False)
    # Plotting loss and confusion matrix for all networks
    plot_results(loss_lists, preds_refs, epoch_num)
```

Code Snippet 7: __main__ run function