ECE 60146 Deep Learning Homework 4

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

February 21, 2023

1 Introduction

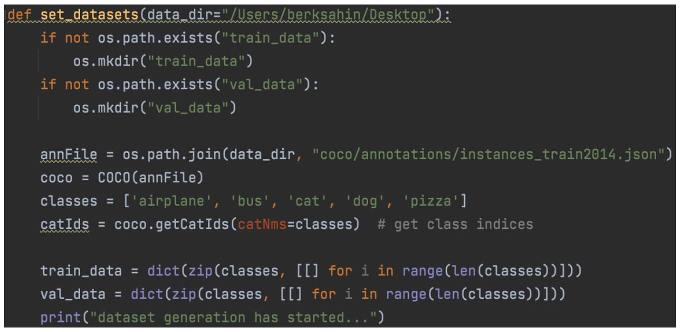
In this homework, there are two main goals: getting familiar with Microsoft Common Objects in COtext (MS-COCO) dataset and creating our own Convolutional Neural Network (CNN) with PyTorch package for image classification. A brief outline of the report is as follows: Section 2 explains the background work for the COCO-API and the sample codes given in DLStudio. Section 3 explains the structure of the dataset designed for the given problem and it explains the important function and classes defined in the source code. In Section 4, the results of the different network architectures are discussed and compared by their training plots and confusion matrices, and questions are answered. To run the whole source code, it is sufficient to run the following code in the terminal "`python3 hw4_MehmetBerkSahin.py --data_dir "/Users/berksahin/Desktop`" at the same directory of the source code. Note that you should do one modification. "data_dir" is the parent folder's path which includes the "coco" folder. The latter one will be elaborated further on the upcoming section.

2 Background Work

To understand the motivations behind the creation of the COCO dataset, I downloaded the paper [1] that introduced the dataset and read it. One of the most important goals of computer vision is the comprehension of visual scenes. This includes identifying which objects are present, localizing or segmenting the objects in 2D and 3D, finding the relationships between the objects, and producing a semantically meaningful description of the scene [1]. The motivation behind COCO dataset is to address the three core research problems, which are as follows accurate 2D localization of objects, contextual relations and reasoning between different objects, and detecting non-iconic views of objects [1].

To curate my dataset, first, I needed to learn COCO-API. So, I downloaded it by writing the following command "`conda install -c conda-forge pycocotools`". Then, I downloaded **2014 Train images** and **2014 Train/Val annotations** datasets from here. As mentioned in the COCO-API GitHub page, I saved **"2014 Train images"** as **"images"** and **"2014 Train/Val annotations"** as **"annotations"** under **"coco"** folder. Basically, COCO-API provides its users to choose images belonging to an arbitrary category/class by filtering with the category's name. This feature was helpful for preparing my dataset.

Lastly, I installed the DLStudio module to play with two CNNs by changing their parameters and see how the classification accuracy got affected by that change. The network in this part is very



```
def set_datasets(data_dir="/Users/berksahin/Desktop"):
    if not os.path.exists("train_data"):
        os.mkdir("train_data")
    if not os.path.exists("val_data"):
        os.mkdir("val_data")

    annFile = os.path.join(data_dir, "coco/annotations/instances_train2014.json")
    coco = COCO(annFile)
    classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
    catIds = coco.getCatIds(catNms=classes)  # get class indices

    train_data = dict(zip(classes, [[] for i in range(len(classes))]))
    val_data = dict(zip(classes, [[] for i in range(len(classes))]))
    print("dataset generation has started...")
```

Figure 1: First part of `set_datasets()`

similar to HW4Net or *Net1*. I played with the hyperparameters such as the number of neurons in fully connected layers, kernel size, padding, and stride to see how these affect the model's performance. I observe that if I increase one of the parameters too much, performance gets worse. That's reasonable because for instance, if the stride is large, then important contextual information in the image may get lost. Similarly, if the kernel size is too much, the algorithm's performance gets worse. I think the reason for this is local information gets lost so if the target object is small, it cannot be learned and classified correctly. On the other hand, if the image is too large, one may use a large stride and kernel size to decrease the computational cost. I also observed that using too small kernel size can decrease the performance. Because I think semantic information between the objects cannot be captured. In short, one should optimize the hyperparameters to obtain the highest accuracy at a reasonable time.

3 Programming Tasks

In this section, first, I explained how I structured my dataset for the given problem in the homework. Then, I explained the important functions and classes in my source code. I did not go through every detail of my source code but you can check it. It is self-explanatory with the comments.

3.1 Creating Your Own Image Classification Dataset

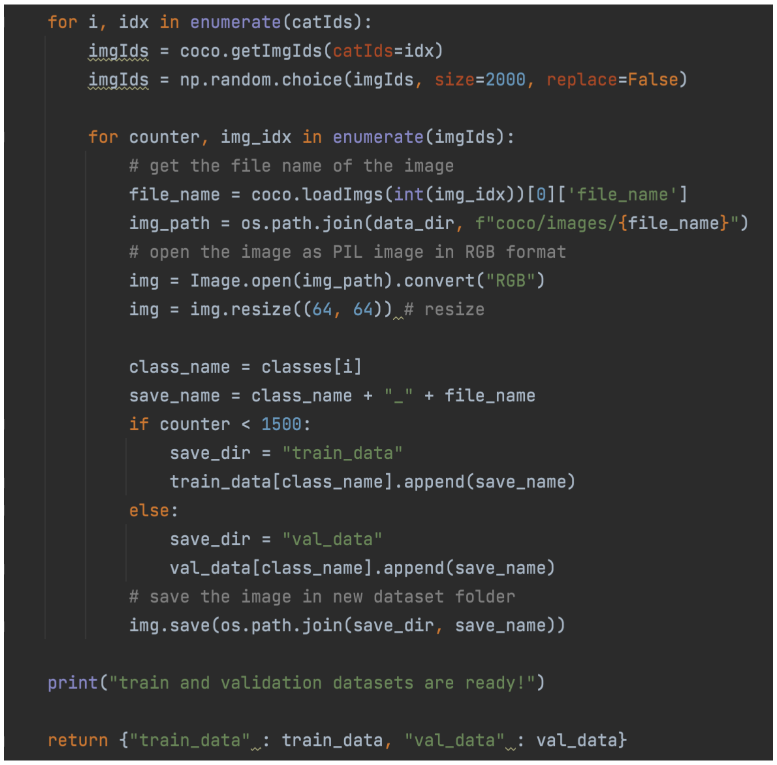
As a first step, I activated my `conda` environment that we created in homework 2 by using "`source activate ECE60146`" command. I downloaded the dataset and set up the COCO API as explained in Section 2. I observed that the "annotations" folder consists of dictionary elements including class labels and URLs. And at the end of the URLs, there are file names that are in the "images" folder, namely **2014 Train images**. With this observation, I created my dataset with `set_datasets()` functiong which I will elaborate on next.

3.1.1 `set_datasets(data_dir)`

It takes the parent folder of the coco folder, which consists of images and annotations folders, as an argument. For example, if coco is under "/Users/berksahin/Desktop", then, data_dir should be equal to that. My dataset consists of 5 classes, which are the following:

`['airplane', 'bus', 'cat', 'dog', 'pizza']`

In the dataset, there are 1500 training and 500 validation images, which are chosen randomly from the COCO dataset according to the labels given in "`instance_train2014.json`", for each class. In



```
for i, idx in enumerate(catIds):
    imgIds = coco.getImgIds(catIds=idx)
    imgIds = np.random.choice(imgIds, size=2000, replace=False)

    for counter, img_idx in enumerate(imgIds):
        # get the file name of the image
        file_name = coco.loadImgs(int(img_idx))[0]['file_name']
        img_path = os.path.join(data_dir, f"coco/images/{file_name}")
        # open the image as PIL image in RGB format
        img = Image.open(img_path).convert("RGB")
        img = img.resize((64, 64)) # resize

        class_name = classes[i]
        save_name = class_name + "_" + file_name
        if counter < 1500:
            save_dir = "train_data"
            train_data[class_name].append(save_name)
        else:
            save_dir = "val_data"
            val_data[class_name].append(save_name)
        # save the image in new dataset folder
        img.save(os.path.join(save_dir, save_name))

print("train and validation datasets are ready!")

return {"train_data": train_data, "val_data": val_data}
```

Figure 2: Second part of `set_datasets()`

total, there are 7.5k training images and 2.5k validation images without any duplicates. All images are resized to 64×64 and saved in ”train_data” and ”val_data” folders in 3-channeled RGB format. I observed some grayscale images in the dataset, and they are expanded to that format by using `convert('RGB')` function of `PIL.Image` class.

The first part of `set_datasets()` is in Figure 1. In the first 4 line, ”train_data” and ”val_data” are created if they do not exist. Then, the coco object is created by using the path of annotations. Class indices are extracted by using COCO API, that is, `getCatIds()` function. And I created a dictionary for each train and validation data whose keys are class names and whose values are lists of file names of images belonging to that class.

In the second part of `set_datasets()`, which can be seen in Figure 2, there are 2 for loops. At the beginning of the outer loop, for each class, I randomly sampled 2000 images from ”images” folder by using COCO API, that is, `getImgIds()`. In the inner loop, for each image, which is randomly selected, I opened them as a PIL image in RGB format (gray scale images are expanded to 3-channeled representations). Then, I resized them and saved them in either ”train_data” or ”val_data” folders with their class names. After the execution of this function is completed, new training data (7.5k images) under ”train_data” directory and validation data (2.5k images) under ”val_data” directory are generated in the same directory as the source code.

I sampled 5 random images for each class from the new dataset that I created. They are plotted in Figure 3. Since they are 64×64, they appear small. And note that some of them are in gray scale because images in COCO dataset are not all colored images. Another observation is that images

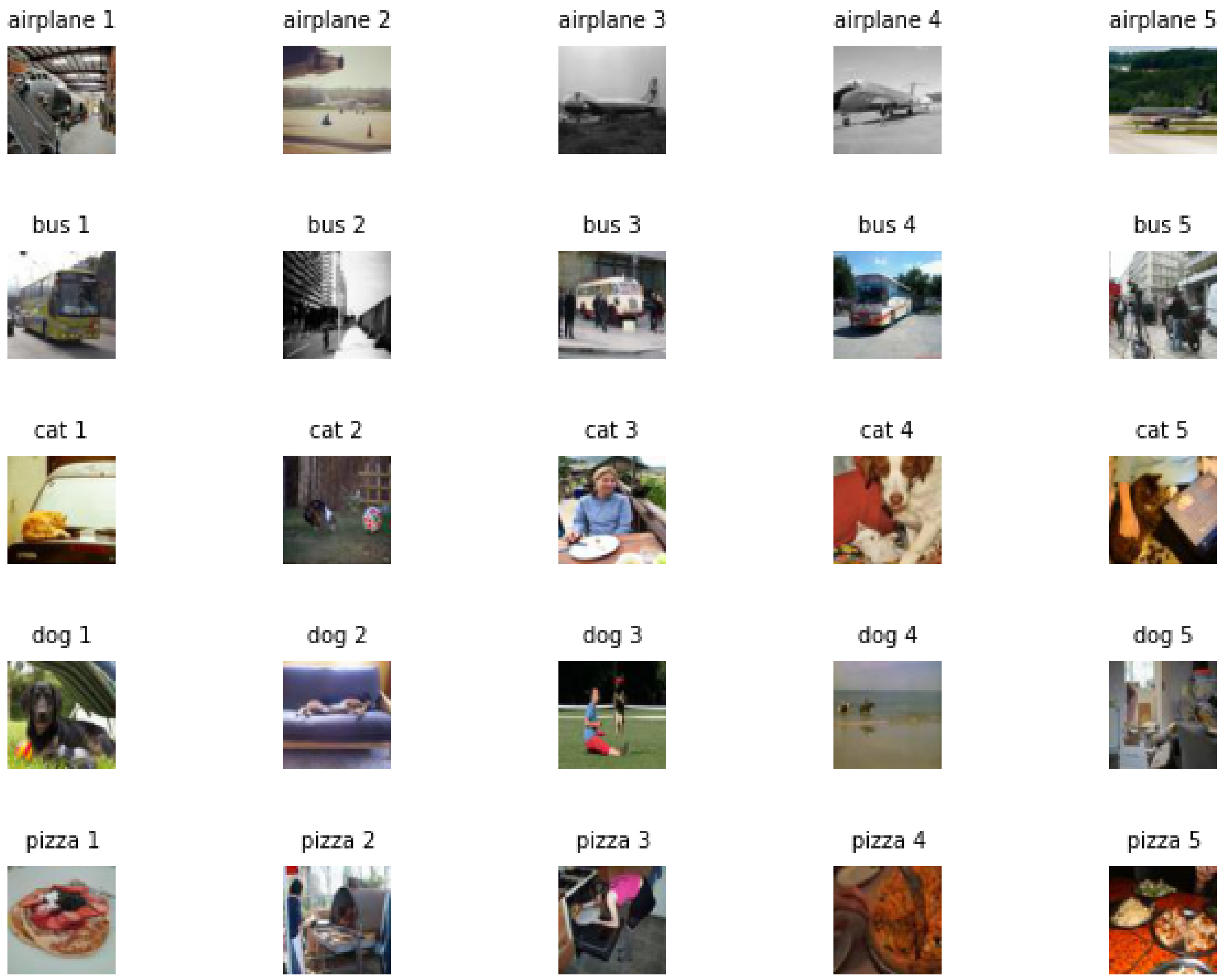
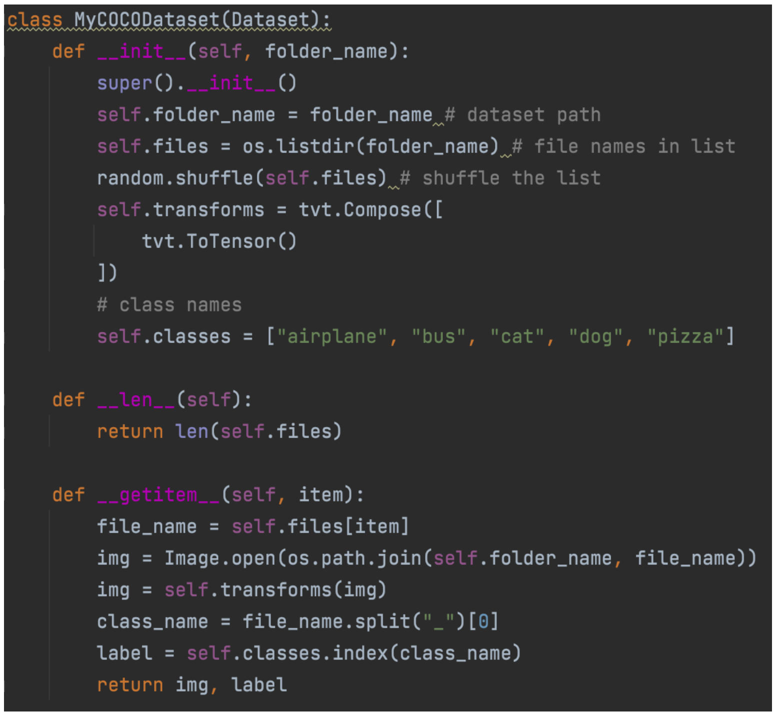


Figure 3: 5 random images from each 5 classes

do not include single objects belonging to the target classes. For instance, "cat 4" includes both cat and dog. Even though the dog occupies much more pixels than cat, the label is "cat". This is inevitable or irreducible noise because there is no reason that neural network or human would prefer one to the other. That kind of noises decrease the validation accuracy of the model. Next, I will explain my custom `Dataset` definition to show you how I utilize train and validation data for training and evaluation of CNNs.

3.1.2 `MyCOCODataset(Dataset)`

I designed this dataset to use the created dataset consisting of 7.5k train and 2.5k validation images, which are 64×64. It is a subclass of `torch.utils.data.Dataset` as most of the custom Datasets in PyTorch. Implementation is very similar to the one we did in homework 2 and it is given in Figure 4. It inherits all the instance variables of its superclass and takes the name of the folder, which consists of either train or validation images. If it is train, then user enters "train_data", else it should be "val_data". Under the initializer, it keeps list of file names belonging to images, it



```python
class MyCOCODataset(Dataset):
    def __init__(self, folder_name):
        super().__init__()
        self.folder_name = folder_name # dataset path
        self.files = os.listdir(folder_name) # file names in list
        random.shuffle(self.files) # shuffle the list
        self.transforms = tvt.Compose([
            tvt.ToTensor()
        ])
        # class names
        self.classes = ["airplane", "bus", "cat", "dog", "pizza"]

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

    def __getitem__(self, item):
        file_name = self.files[item]
        img = Image.open(os.path.join(self.folder_name, file_name))
        img = self.transforms(img)
        class_name = file_name.split("_")[0]
        label = self.classes.index(class_name)
        return img, label
```

Figure 4: Implementation of custom dataset `MyCOCODataset(Dataset)`

has rescaling transformation and class names in a list. Whenever an item is popped out, image is converted from PIL format to PyTorch tensor and entries are rescaled to $[0, 1]$. Then, image tensor with its class index, which is regarded as label, are returned as an item from the dataset. For instance, the label of "airplane" is 0 and "dog" is 3. This fact will be used later.

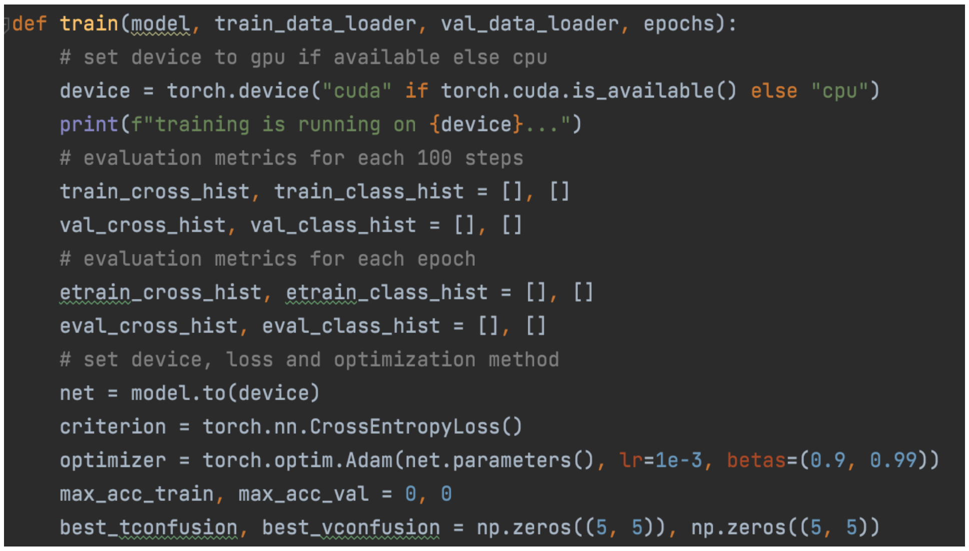
3.2 Image Classification using CNNs - Training and Validation

In this part, I explained the source code, discuss the results and answered the given questions in the homework. I will go over the source code by first discussing the main training loop, which is used for each CNN tasks, then I will explain the CNN PyTorch class definitions for each CNN tasks and discuss their results with plots and confusion matrices. Since the main training loop is too big to put at once, I will divide it into parts and explain them seperately.

3.2.1 `train()`

I took the training loop provided in homework 4 and modified it to obtain classification accuracy and cross-entropy losses for train and validation data at each 100 steps and each epochs. Moreover, I initialized confusion matrices for both training and validation data and update it at each iteration of the DataLoader. All these evaluation metrics are saved in dictionary and returned to be plotted later.

The first part of the function can be found in Figure 5. It takes 4 parameters. **`model`** is an instance of my CNN class definitions, which are subclass of `torch.nn.Module`. `train_data_loader` and `val_data_loader` parameters are `torch.utils.data.DataLoader` instances, which are combined with my custom Dataset definition given in Figure 4. Lastly, `epochs` is how many times the training goes over the entire data.



```
def train(model, train_data_loader, val_data_loader, epochs):
    # set device to gpu if available else cpu
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"training is running on {device}...")
    # evaluation metrics for each 100 steps
    train_cross_hist, train_class_hist = [], []
    val_cross_hist, val_class_hist = [], []
    # evaluation metrics for each epoch
    etrain_cross_hist, etrain_class_hist = [], []
    eval_cross_hist, eval_class_hist = [], []
    # set device, loss and optimization method
    net = model.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.99))
    max_acc_train, max_acc_val = 0, 0
    best_tconfusion, best_vconfusion = np.zeros((5, 5)), np.zeros((5, 5))
```

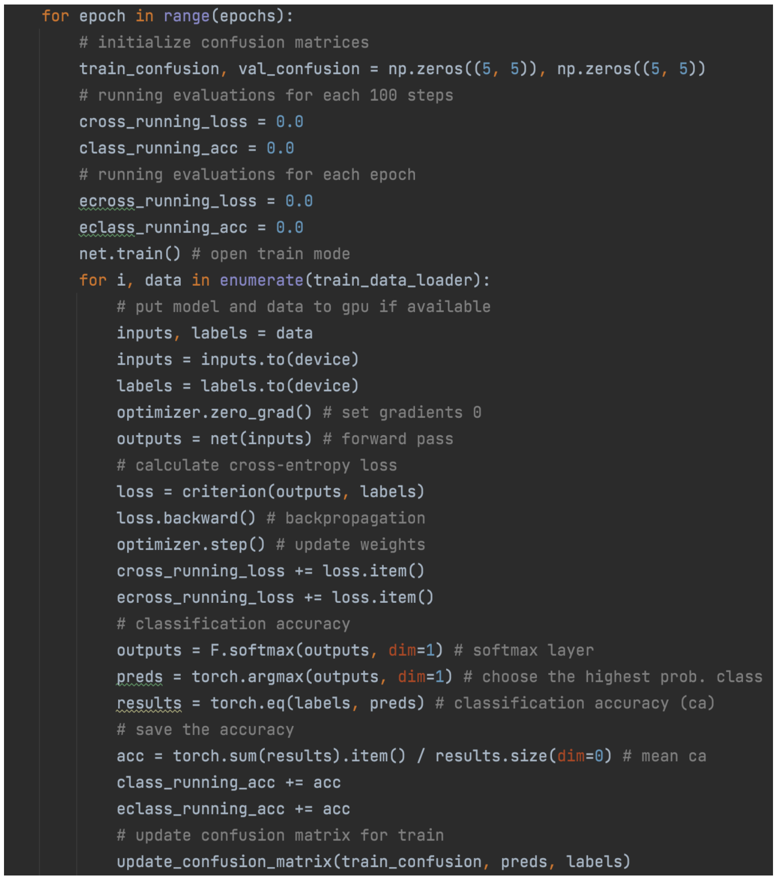
Figure 5: First part of `train()`

In the first line, device is set to GPU if it is available else it is set to CPU. Doing this prevents the errors if GPU is not available. Then, excluding the comment and print statements, the following 4 lines are initializations of loss and accuracy histories of train and validation sets, which are recorded at each 100 steps and each epoch. In the next lines, CNN model is loaded to the available device (GPU if available) to perform tensor operations fastly. And loss function (`criterion`) and optimization method (`optimizer`) are set to Cross-entropy loss and Adam optimizer respectively. Lastly, cross-entropy loss is one of the important evaluation metrics for classification tasks. For one sample, it is defined as follows:

$$\rho(y_i, \hat{y}_i) = -\sum_i y_i \log(\hat{y}_i) \tag{1}$$

where y_i and $\hat{y}_i$ are the i^{th} entries of y, $\hat{y} \in \mathbb{R}^n$. And it can be easily generalized to the entire dataset. y and $\hat{y}$ are one-hot-encoded ground truth and prediction probability vectors respectively. As you may notice, there is only one non-zero entry in y so the purpose is to make the probability of target class 1 in prediction vector. From another perspective, KL-divergence and cross-entropy are positively correlated so minimizing cross-entropy minimizes the dissimilarity between the probability distributions of ground truths and predictions. Lastly, in the last line of 5, the initialization of confusion matrices for train and validation data were done. Since there are 5 different classes, their size is 5×5. Note that they keep the best performance of the model for the train and validation data. Instead of doing that, I may also save the best model and asses the validation performance later.

Following the first part in Figure 5, there are two loops, which can be seen in Figure 6. Outer loop iterates through epochs and the inner loops iterates through the batches for each epoch. First 5 line of the outer loop initializes the variables, which keep loss and accuracy values for each mini-batch. In the inner loop, data is extracted and loaded into GPU if available. Then, gradients are set to 0. If `zero_grad()` was not used, then gradients from previous steps would accumulate and result in wrong updates. Then, forward pass is done. Loss value is calculated with the criterion (cross-entropy in our problem), which is argument of `train()`. Then, backpropagation and updates were



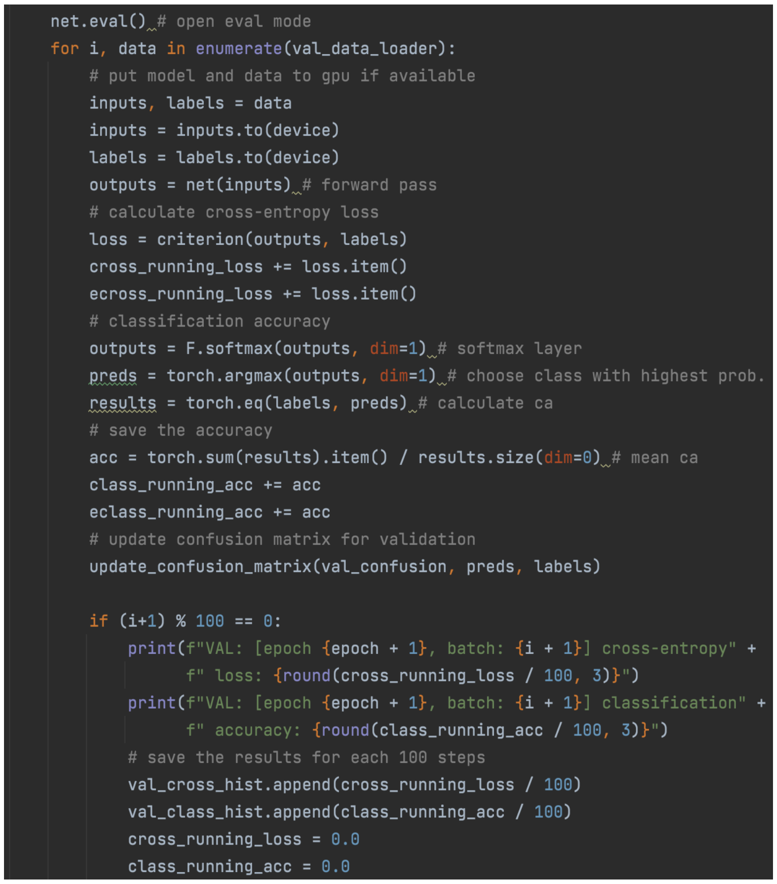
```python
for epoch in range(epochs):
    # initialize confusion matrices
    train_confusion, val_confusion = np.zeros((5, 5)), np.zeros((5, 5))
    # running evaluations for each 100 steps
    cross_running_loss = 0.0
    class_running_acc = 0.0
    # running evaluations for each epoch
    ecross_running_loss = 0.0
    eclass_running_acc = 0.0
    net.train() # open train mode
    for i, data in enumerate(train_data_loader):
        # put model and data to gpu if available
        inputs, labels = data
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer.zero_grad() # set gradients 0
        outputs = net(inputs) # forward pass
        # calculate cross-entropy loss
        loss = criterion(outputs, labels)
        loss.backward() # backpropagation
        optimizer.step() # update weights
        cross_running_loss += loss.item()
        ecross_running_loss += loss.item()
        # classification accuracy
        outputs = F.softmax(outputs, dim=1) # softmax layer
        preds = torch.argmax(outputs, dim=1) # choose the highest prob. class
        results = torch.eq(labels, preds) # classification accuracy (ca)
        # save the accuracy
        acc = torch.sum(results).item() / results.size(dim=0) # mean ca
        class_running_acc += acc
        eclass_running_acc += acc
        # update confusion matrix for train
        update_confusion_matrix(train_confusion, preds, labels)
```

Figure 6: Second part of `train()`

done by using `backward()` and `step()` functions. Cross-entropy loss is calculated and classification accuracy is calculated by comparing the predicted class with the highest probability and ground truths (`torch.eq()`). Lastly, classification accuracy is saved and confusion matrix is updated by using `update_confusion_matrix()` function, which will be elaborated later.

This inner loop is for training because it iterates through training DataLoader and at the beginning of the inner loop, model is set to training mode (`net.train()`). There are few more lines in the training inner loop, which are not seen in Figure 6. They basically print the loss and accuracy values for each 100 steps and save the loss and accuracy to lists for each 100 steps and epochs. Lastly, they reset the running losses and update the best confusion matrix with respect to training classification accuracy. For more detail, you can check the source code.

The third part of `train()` function, which can be seen in Figure 7, consists of the validation loop. I put the validation loop under the same loop with the training because I wanted to compare the results of validation and training for each epoch. Validation loop is very similar to the training loop except it is in evaluation mode. Thus, before the loop, model is set to evaluation mode by `net.eval()`. Furthermore, there is neither gradient calculation nor weight updates so `backward()` and `step()` functions are not used here. Other steps are the same with the training loop. In the last part of `train()` given in Figure 8, I wanted to point out the returned objects. Function basicall returns a dictionary of dictionaries. User first decides whether s/he wants to access the evaluation metrics calculated at each epoch or 100 steps or the confusion matrix. Then, s/he can choose to



```python
net.eval() # open eval mode
for i, data in enumerate(val_data_loader):
    # put model and data to gpu if available
    inputs, labels = data
    inputs = inputs.to(device)
    labels = labels.to(device)
    outputs = net(inputs) # forward pass
    # calculate cross-entropy loss
    loss = criterion(outputs, labels)
    cross_running_loss += loss.item()
    ecross_running_loss += loss.item()
    # classification accuracy
    outputs = F.softmax(outputs, dim=1) # softmax layer
    preds = torch.argmax(outputs, dim=1) # choose class with highest prob.
    results = torch.eq(labels, preds) # calculate ca
    # save the accuracy
    acc = torch.sum(results).item() / results.size(dim=0) # mean ca
    class_running_acc += acc
    eclass_running_acc += acc
    # update confusion matrix for validation
    update_confusion_matrix(val_confusion, preds, labels)

    if (i+1) % 100 == 0:
        print(f"VAL: [epoch {epoch + 1}, batch: {i + 1}] cross-entropy" +
              f" loss: {round(cross_running_loss / 100, 3)}")
        print(f"VAL: [epoch {epoch + 1}, batch: {i + 1}] classification" +
              f" accuracy: {round(class_running_acc / 100, 3)}")
        # save the results for each 100 steps
        val_cross_hist.append(cross_running_loss / 100)
        val_class_hist.append(class_running_acc / 100)
        cross_running_loss = 0.0
        class_running_acc = 0.0
```

Figure 7: Third part of `train()`

access train or validation loss or accuracy. For example, if the returned dictionary is assigned to the variable `result`, then user can access the training cross-entropy loss for each epoch as follows:

`result["epoch"]["train_cross_entropy"]`,

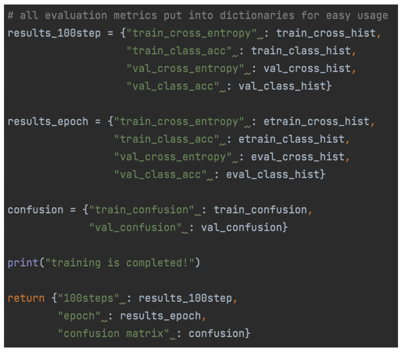
and the loss will be returned in Python list format. Next, I want to explain how I updated and plotted the confusion matrices.

3.2.2 `update_confusion_matrix`

It takes three arguments, confusion matrix to be updated, predicted labels and true labels respectively. The update rule in Figure 9 is very simple and as follows: if the predicted label is i and true label is j where $i, j \in \{0, 1, 2, 3, 4\}$, then j^{th} row of i^{th} column of the confusion matrix is incremented by one. This is done for each sample in the batch and the updated confusion matrix is returned. In the following subsection, you can see how they are plotted.

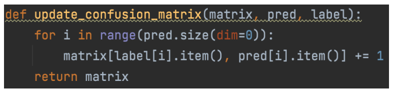
3.2.3 `plot_confusion_matrix()`

This function is designed to plot the confusion matrix of train and validation datasets. Its implementation is given in Figure 10. It takes two arguments, which are the network name and results. From the results it extracts the confusion matrices for train and validation sets and they are plotted by using `seaborn.heatmap()` function. In the code, `sns` stands for `seaborn.` I will not explain



```python
# all evaluation metrics put into dictionaries for easy usage
results_100step = {"train_cross_entropy" : train_cross_hist,
                   "train_class_acc" : train_class_hist,
                   "val_cross_entropy" : val_cross_hist,
                   "val_class_acc" : val_class_hist}

results_epoch = {"train_cross_entropy" : etrain_cross_hist,
                 "train_class_acc" : etrain_class_hist,
                 "val_cross_entropy" : eval_cross_hist,
                 "val_class_acc" : eval_class_hist}

confusion = {"train_confusion" : train_confusion,
             "val_confusion" : val_confusion}

print("training is completed!")

return {"100steps" : results_100step,
        "epoch" : results_epoch,
        "confusion matrix" : confusion}
```

Figure 8: Fourth part of `train()`



```python
def update_confusion_matrix(matrix, pred, label):
    for i in range(pred.size(dim=0)):
        matrix[label[i].item(), pred[i].item()] += 1
    return matrix
```

Figure 9: Implementation of `update_confusion_matrix`

how `matplotlib.pyplot` functions work because I already did them in previous homeworks so I will skip the functions for plotting the loss and accuracy. For more detail, you can check the source code and comments. In the next section, I will explain my class definitions for CNN tasks but before getting into that. I want to give the formula of how I calculated the number of neurons in the input of the fully connected layers.

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding[0] - dilation[0] \times (kernel\_size[0] - 1) - 1}{stride[0]} + 1 \right\rfloor \tag{2}$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times padding[1] - dilation[1] \times (kernel\_size[1] - 1) - 1}{stride[1]} + 1 \right\rfloor \tag{3}$$

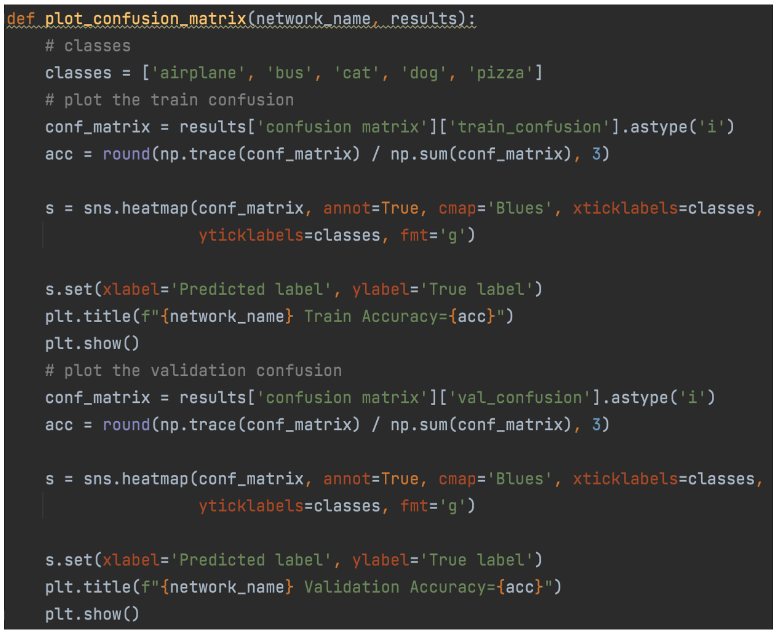
These formulas are taken from PyTorch documentation. Although their derivation is not hard, I do not do them here because that's not the purpose of this homework.

3.2.4 `HW4Net(nn.Module)`

This network is given in the homework and we are asked to find `XXXX` and `XX` values. The former is the number of neurons at the first fully connected layer. There are two ways of finding this: using the equations in 2 and 3 for each layer or running forward pass up to the input of the first fully connected layer, and then by flattening the output, one can check its dimensionality. I did both and verify the algorithm with random tensors by performing complete forward pass. I found that `XXXX=5408`. Furthermore, since there are 5 classes in our problem, there should be 5 neurons at the output of the fully connected layers. Because after passing them through softmax function, the outputs will represent the probabilities of the classes. Thus, `XX=5`. Full implementation of `HW4Net` or *Net1* is given in Figure 11a.

3.2.5 `Net2(HW4Net)`

The implementation of a CNN for **CNN Task 2** is given in Figure 11b. The only difference between this and *Net1* is padding of one is added to the convolutional layers. Thus, I designed it as a subclass of *Net1* and only modified the initializer of the class. That is, I only added padding



```python
def plot_confusion_matrix(network_name, results):
    # classes
    classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
    # plot the train confusion
    conf_matrix = results['confusion matrix']['train_confusion'].astype('i')
    acc = round(np.trace(conf_matrix) / np.sum(conf_matrix), 3)

    s = sns.heatmap(conf_matrix, annot=True, cmap='Blues', xticklabels=classes,
                    yticklabels=classes, fmt='g')

    s.set(xlabel='Predicted label', ylabel='True label')
    plt.title(f"{network_name} Train Accuracy={acc}")
    plt.show()
    # plot the validation confusion
    conf_matrix = results['confusion matrix']['val_confusion'].astype('i')
    acc = round(np.trace(conf_matrix) / np.sum(conf_matrix), 3)

    s = sns.heatmap(conf_matrix, annot=True, cmap='Blues', xticklabels=classes,
                    yticklabels=classes, fmt='g')

    s.set(xlabel='Predicted label', ylabel='True label')
    plt.title(f"{network_name} Validation Accuracy={acc}")
    plt.show()
```

Figure 10: Implementation of `plot_confusion_matrix()`

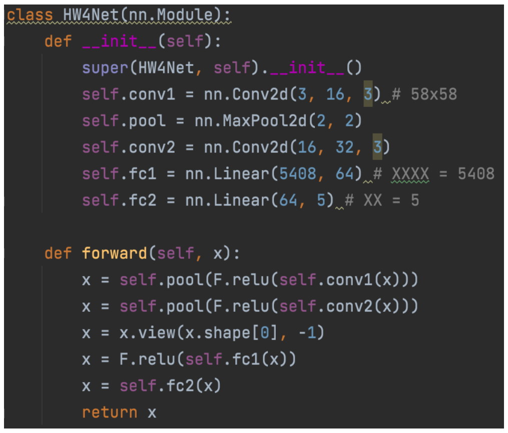
of one to the convolutional layers. As the forward pass is exactly the same, I did not change that part.

3.2.6 `Net3(HW4Net)`

Net1 and *Net2* architectures are very shallow networks. In **CNN Task 3**, we are asked to design a deeper network by modifying `HW4Net` class. I added a chain of 10 extra convolutional layers between the second conv layer and the first linear layer. Each convolution layer has 32 in-channels and 32 out-channels, a kernel size of 3, and padding of 1. Chain of convolutional layers are kept in `self.conv_chain` as a list. And the number of neurons for the first linear layer is updated to `XXXX=5408`. In `forward()` function, they are taken out from the list and used in the forward pass under the for a loop. The other parts are the same as *Net1* and *Net2*. Lastly, for this architecture, I did not use the formulas given in 2 and 3 to find `XXXX`. Because doing the same calculations for each layer would be cumbersome but it is possible to automatize this process as done in DLStudio. Instead of using the formulas, I checked the dimensionality of the output of the chain of convolutions and assign it to the number of neurons in the linear layer.

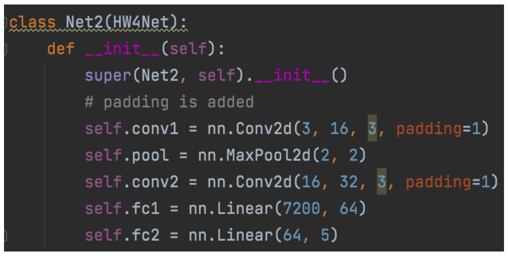
4 Results

In this section, I put the results of the experiments and discussed them. I answered the questions given in the homework. Although I did not do that as it was not asked in the assignment, it is very easy to run the same experiments, which are **CNN Task 1, 2, and 3**, with various augmentations such as rotation, flip, and random perspective to increase the performance. To achieve that, one can simply add PyTorch objects defined for these transformations into the `tvt.Compose()` initialized in custom dataset class definition, Figure 4. In all experiments, I used cross-entropy loss and mean classification accuracy metrics to evaluate the performance of the model. The latter is the ratio of the correct predictions to the number of predictions. Adam optimizer is used with SGD. Learning



```python
class HW4Net(nn.Module):
    def __init__(self):
        super(HW4Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3) # 58x58
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.fc1 = nn.Linear(5408, 64) # XXXX = 5408
        self.fc2 = nn.Linear(64, 5) # XX = 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
```

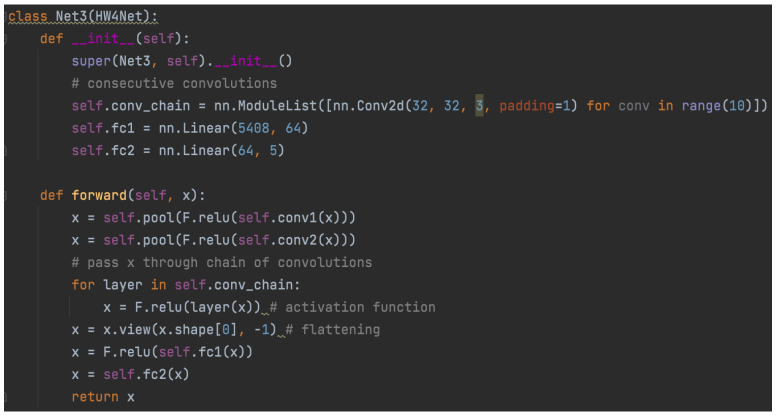
(a) *Net1*



```python
class Net2(HW4Net):
    def __init__(self):
        super(Net2, self).__init__()
        # padding is added
        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(7200, 64)
        self.fc2 = nn.Linear(64, 5)
```

(b) *Net2*

Figure 11: Implementation of *Net1* and *Net2*

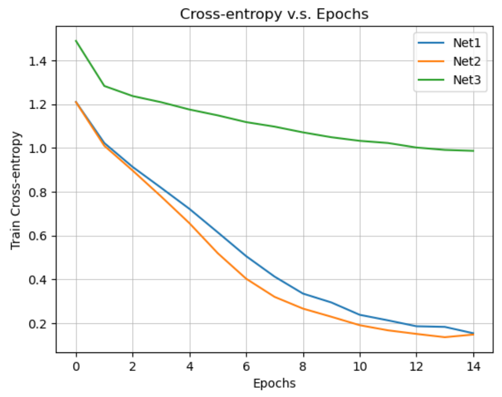


```python
class Net3(HW4Net):
    def __init__(self):
        super(Net3, self).__init__()
        # consecutive convolutions
        self.conv_chain = nn.ModuleList([nn.Conv2d(32, 32, 3, padding=1) for conv in range(10)])
        self.fc1 = nn.Linear(5408, 64)
        self.fc2 = nn.Linear(64, 5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        # pass x through chain of convolutions
        for layer in self.conv_chain:
            x = F.relu(layer(x)) # activation function
        x = x.view(x.shape[0], -1) # flattening
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

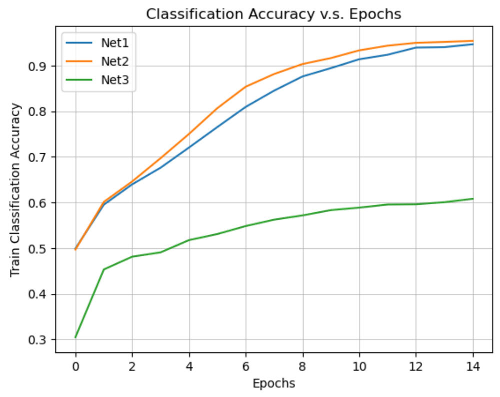
Figure 12: Implementation of *Net3*

rate, β_1 and β_2 are chosen to be 10^{-3}, 0.9 and 0.99 respectively. The batch size is 4 and the number of epochs for training is 15. Each network's performance is evaluated based on its best performance on the validation dataset in 15 epochs. This is reasonable because one can save the best model to use later on another dataset during the training. Lastly, the reason the model is evaluated on the validation set is that it consists of the examples that the model does not see and get trained on.

I saved the cross-entropy loss and classification accuracy at every 100 iterations of the DataLoader and at every epoch. Learning curves for the cross-entropy loss and classification accuracy for each epoch can be found in Figure 13. The first observation is that *Net 1*'s and *Net 2*'s performances are very close to each other. Because the only difference between the two models is *Net 2* has padding of one. That is, it adds zero pixels to all sides of the image. This is done to prevent image size from reducing too much. The second observation is the training performance of the models is $Net2 > Net1 > Net3$ because *Net2* has the lowest cross-entropy in Figure 13a and has the highest accuracy in Figure 13b. It is followed by *Net2* and *Net3*. However, one should note that high training performance does not always mean it will generalize well and achieve a good test performance. It may overfit and memorize the samples. Another noteworthy thing is that although *Net 3* is the deepest network, its training performance is the worst. This is contrary to the intuition that deeper networks are more capable of learning the data. In Deep Learning



(a) cross-entropy v.s. epochs

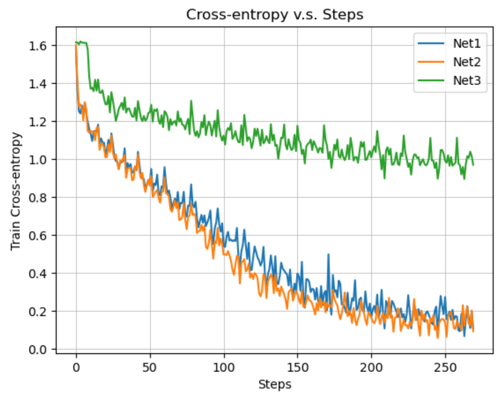


(b) classification accuracy v.s. epochs

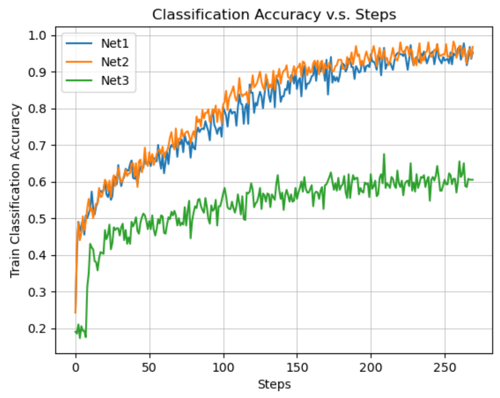
Figure 13: Learning curves of *Net1*, *Net2*, and *Net3* for each epoch

literature, this problem is referred to as the vanishing gradient problem. It happens when the weight update approaches 0 and prevents the learning of the model. For example, if the network is deep and we want to make an update at the first layer, if the updates between the first layer and the output layer are small, their multiplication will be very close to 0. Hence, the first layer does not learn from its errors. To prevent this, one may add residual connections from the previous layers to the next layers [2] or batch normalization layers can be added after every convolutional layer [3].

Next, I want to discuss the learning curves for cross-entropy loss and classification accuracy saved at every 100 iterations of the DataLoader. Training curves for that case can be seen in Figure 14. These plots are noisier than Figure 13 because points represent the mean loss or accuracy at every 100 batches whereas in Figure 13, they represent the mean loss at every epoch, which is roughly 1800 batches. Next, I will discuss the confusion matrices for training and validation sets.



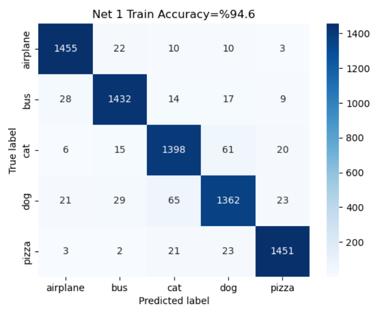
(a) cross-entropy v.s. steps



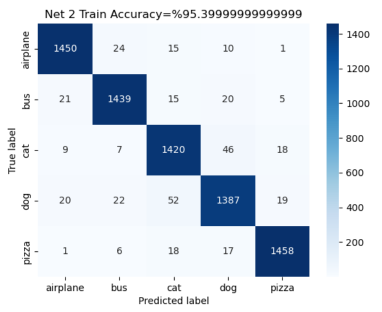
(b) classification accuracy v.s. steps

Figure 14: Learning curves of *Net1*, *Net2*, and *Net3* for each step (1 step = 100 iteration)

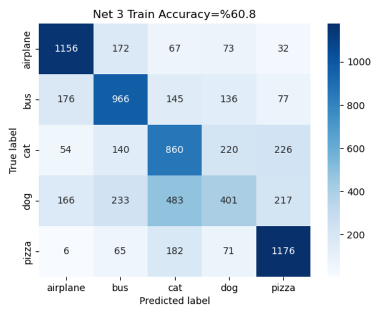
You can check the confusion matrices for the training set in Figure 15. Note that these results are the best results of the models over 15 epochs. In a confusion matrix, the ideal case is having a matrix of diagonal elements equal to zero and non-diagonal elements equal to zero. This corresponds to all predictions being correct. In Figure 15, the order of training accuracy is the following $Net2 > Net1 > Net3$. One observation is that in all three confusion matrices, models did the most mistake in the classification of dog and cat. Especially, *Net3* did a lot of wrong predictions between them. This is expected because as we saw in Figure 3, some images include both cat and dog images, which makes the task quite challenging.



(a) *Net1* confusion matrix for training



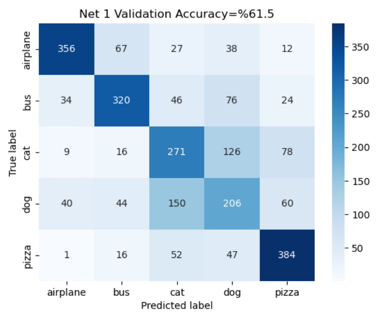
(b) *Net2* confusion matrix for training



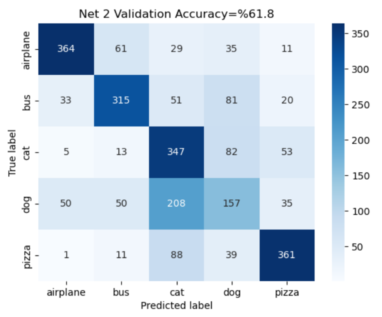
(c) *Net3* confusion matrix for training

Figure 15: Confusion matrices of *Net1*, *Net2*, and *Net3* for training

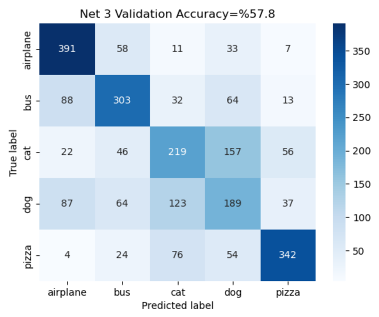
In Figure 16, you can check the confusion matrices of the models for validation sets. The order of the performance of the models remains the same. Different from Figure 15, the difficulty in distinguishing cats and dogs becomes more apparent on the validation set. Also, every model's performance decreased because models are evaluated on unseen samples, which they were not trained on. Thus, they can encounter strange or difficult examples that they have not seen before. Moreover, it seems that the problem of vanishing gradients appears here, and although *Net3* is the deepest model, it has the worst performance. Another reason that validation performances are low can be as we used 64×64 downscaled images, which decreases the resolution tremendously. For instance, if we used 250×250, the validation accuracy could be higher than that. Lastly, I want to point out that the order of the models and their losses and accuracies can vary because I did not use random seed for PyTorch. Meaning that, there are still randomness for the initialization of weights of the layers. Although they may vary, there will not be any significant change on the performance of the models.



(a) *Net1* confusion matrix for validation



(b) *Net2* confusion matrix for validation



(c) *Net3* confusion matrix for validation

Figure 16: Confusion matrices of *Net1*, *Net2*, and *Net3* for validation

Next, I will answer the questions in the homework problem one by one. Although I already answered some of them, I will quickly go over them for easy reference.

Q1: Does adding padding to the convolutional layers make a difference in classification performance?

A: Yes, it increased the classification accuracy of training performance and validation performance by %0.8 and %0.4. I think this happened because thanks to the padding, we used more pixels in the convolution operation, therefore the network learns from the edges more.

Q2: As you may have known, naively chaining a large number of layers can result in difficulties in training. This phenomenon is often referred to as vanishing gradient. Do you observe something like that in *Net3*?

A: Yes! I observed vanishing gradient problem indeed. If you look at Figure 13 and Figure 14, you will see that after some time, losses started to decrease very slowly compared to *Net1* and *Net2*. The same is true for accuracy as well. As we increased the number of layers naively, that is, without taking any precautions such as batch normalization or residual connection, gradients get very small and updates approach 0. Because of this, *Net3* could not learn and generalize well so it exhibits poor validation performance.

Q3: Compare the classification results by all three networks, which CNN do you think is the best performer?

A: As I mentioned previously, comparing the classification performances of three CNN on both train and validation sets, I think *Net2* is the best performer. Then, *Net2* comes and *Net3* is the worst performer due to vanishing gradients. In fact, I think to choose the best performer, it is sufficient to look at the validation set (assuming there is no test set) without looking at the training set. Because, as I mentioned, the generalization performance of the model is important and it is measured by observing how the network performs with unseen samples, which are not used in the training.

Q4: By observing your confusion matrices, which class or classes do you think are more difficult to correctly differentiate and why?

A: As I explained in the discussion part, looking at the confusion matrices, I think cat and dog classes are the most difficult to differentiate. Because if we consider the 2×2 submatrix in validation confusion matrices for dog and cat classes, it has large non-diagonal elements and they are darker (or larger in quantity) than the other non-diagonal elements of the confusion matrix. This is true for the confusion matrices of *Net1*, *Net2*, and *Net3*. Additionally, *Net3* has some difficulties to differentiate airplane and bus but it is not as worse as the dog and cat problems. The problem with the cat and dog classification is that there are some images that are labeled as a cat but also include a dog (i.e., cat image 4 in Figure 3).

Q5: What is one thing that you propose to make the classification performance better?

A: To increase the classification performance of *Net3*, residual connections can be added or batch normalization can be used between convolutional layers. To increase the classification performance in general, we can increase the resolution of the images because we lost too much information when we downscaled the images to 64×64. We can increase the size of the training dataset for every class equally. We can add noise to the samples to regularize the learning for *Net1* and *Net2* because they overfit the data. The gap between the training and validation performance is too large. As a regularization, we can apply the dropout technique. We may increase the number of fully connected layers. Lastly, we may do Xavier initialization for the weights of each layer to reduce the bias.

5 Lessons Learned

In this homework, I learned how to create my classification dataset based one the images and annotations taken from COCO dataset. And I achieved this by using COCO API. I practiced my skills on creating my custom dataset and dataloader which we learned in previous homeworks. Moreover, I learned how to design a main training loop, which consists of forward pass, backpropagation, optimizer, loss criterion, loading the model and data to GPU if available, and saving the loss and accuracy metrics to plot the learning curve. I also learned how to construct and plot confusion matrix, which is crucial for evaluating the performance of a classifier. I also learned using different modes ('same' or 'valid') of convolutional layers and observed how padding, stride, kernel size change the size of the images. Lastly, I learned how to calculate the number of neurons at the first fully connected layer, which is after the last convolutional layer.

References

[1] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

```python
# author: Mehmet Berk Sahin
# import necessary packages
import argparse
import random
import torch
from pycocotools.coco import COCO
import numpy as np
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as tvt
import os
import pickle
from PIL import Image
import seaborn as sns
import matplotlib.pyplot as plt

class MyCOCODataset(Dataset):
    def __init__(self, folder_name, aug=False):
        super().__init__()
        self.folder_name = folder_name # dataset path
        self.files = os.listdir(folder_name) # file names in list
        random.shuffle(self.files) # shuffle the list
        # Later one can add augmentations to this pipeline easily
        self.transforms = tvt.Compose([
            tvt.ToTensor()
        ])
        # class names
        self.classes = ["airplane", "bus", "cat", "dog", "pizza"]

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

    def __getitem__(self, item):
        file_name = self.files[item]
        img = Image.open(os.path.join(self.folder_name, file_name)) # load image
        img = self.transforms(img) # convert it to tensor with normalization
        class_name = file_name.split("_")[0] # get the index of label
        label = self.classes.index(class_name) # get label
        return img, label

class HW4Net(nn.Module):
    def __init__(self):
        super(HW4Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3) # 58x58
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.fc1 = nn.Linear(5408, 64) # XXXX = 5408
        self.fc2 = nn.Linear(64, 5) # XX = 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

class Net2(HW4Net):
    def __init__(self):
        super(Net2, self).__init__()
        # padding is added
        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(7200, 64)
        self.fc2 = nn.Linear(64, 5)
```

```python
class Net3(HW4Net):
    def __init__(self):
        super(Net3, self).__init__()
        # consecutive convolutions
        self.conv_chain = nn.ModuleList([nn.Conv2d(32, 32, 3, padding=1) for conv in
range(10)])
        # fully connected classifier layers
        self.fc1 = nn.Linear(5408, 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)))
        # pass x through chain of convolutions
        for layer in self.conv_chain:
            x = F.relu(layer(x)) # activation function
        x = x.view(x.shape[0], -1) # flattening
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def update_confusion_matrix(matrix, pred, label):
    for i in range(pred.size(dim=0)):
        matrix[label[i].item(), pred[i].item()] += 1
    return matrix

def set_datasets(data_dir="/Users/berksahin/Desktop"):
    # create folders
    if not os.path.exists("train_data"):
        os.mkdir("train_data")
    if not os.path.exists("val_data"):
        os.mkdir("val_data")

    annFile = os.path.join(data_dir, "coco/annotations/instances_train2014.json")
    coco = COCO(annFile)
    # classes for the problem in hw4
    classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
    catIds = coco.getCatIds(catNms=classes)  # get class indices
    # keys: class names, values: list of files names
    train_data = dict(zip(classes, [[] for i in range(len(classes))]))
    val_data = dict(zip(classes, [[] for i in range(len(classes))]))
    print("dataset generation has started...")
    for i, idx in enumerate(catIds):
        imgIds = coco.getImgIds(catIds=idx)
        imgIds = np.random.choice(imgIds, size=2000, replace=False)

        for counter, img_idx in enumerate(imgIds):
            # get the file name of the image
            file_name = coco.loadImgs(int(img_idx))[0]['file_name']
            img_path = os.path.join(data_dir, f"coco/images/{file_name}")
            # open the image as PIL image in RGB format
            img = Image.open(img_path).convert("RGB")
            img = img.resize((64, 64)) # resize

            class_name = classes[i]
            save_name = class_name + "_" + file_name
            if counter < 1500:
                save_dir = "train_data"
                train_data[class_name].append(save_name)
            else:
                save_dir = "val_data"
                val_data[class_name].append(save_name)
            # save the image in new dataset folder
            img.save(os.path.join(save_dir, save_name))

    print("train and validation datasets are ready!")
    return {"train_data" : train_data, "val_data" : val_data}

def train(model, train_data_loader, val_data_loader, epochs):
    # set device to gpu if available else cpu
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"training is running on {device}...")
    # evaluation metrics for each 100 steps
    train_cross_hist, train_class_hist = [], []
    val_cross_hist, val_class_hist = [], []
    # evaluation metrics for each epoch
    etrain_cross_hist, etrain_class_hist = [], []
    eval_cross_hist, eval_class_hist = [], []
    # set device, loss and optimization method
    net = model.to(device)
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, betas=(0.9, 0.99))
    max_acc_train, max_acc_val = 0, 0
    best_tconfusion, best_vconfusion = np.zeros((5, 5)), np.zeros((5, 5))
    for epoch in range(epochs):
        # initialize confusion matrices
        train_confusion, val_confusion = np.zeros((5, 5)), np.zeros((5, 5))
        # running evaluations for each 100 steps
        cross_running_loss = 0.0
        class_running_acc = 0.0
        # running evaluations for each epoch
        ecross_running_loss = 0.0
        eclass_running_acc = 0.0
        net.train() # open train mode
        for i, data in enumerate(train_data_loader):
            # put model and data to gpu if available
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad() # set gradients 0
            outputs = net(inputs) # forward pass
            # calculate cross-entropy loss
            loss = criterion(outputs, labels)
            loss.backward() # backpropagation
            optimizer.step() # update weights
            cross_running_loss += loss.item()
            ecross_running_loss += loss.item()
            # classification accuracy
            outputs = F.softmax(outputs, dim=1) # softmax layer
            preds = torch.argmax(outputs, dim=1) # choose the highest prob. class
            results = torch.eq(labels, preds) # classification accuracy (ca)
            # save the accuracy
            acc = torch.sum(results).item() / results.size(dim=0) # mean ca
            class_running_acc += acc
            eclass_running_acc += acc
            # update confusion matrix for train
            update_confusion_matrix(train_confusion, preds, labels)

            if (i+1) % 100 == 0:
                print(f"TRAIN: [epoch {epoch + 1}, batch: {i + 1}] cross-entropy" +
                      f" loss: {round(cross_running_loss / 100, 3)}")
                print(f"TRAIN: [epoch {epoch + 1}, batch: {i + 1}] classification" +
                      f" accuracy: {round(class_running_acc / 100, 3)}")
                # save the results for each 100 steps
                train_cross_hist.append(cross_running_loss / 100)
                train_class_hist.append(class_running_acc / 100)
                cross_running_loss = 0.0
                class_running_acc = 0.0

        mean_acc = eclass_running_acc / len(train_data_loader)
        # save the results for each epoch
        etrain_cross_hist.append(ecross_running_loss / len(train_data_loader))
        etrain_class_hist.append(mean_acc)
        # save the best performance
        if mean_acc > max_acc_train:
            best_tconfusion = train_confusion
            max_acc_train = mean_acc

        # reset epoch evaluations
        ecross_running_loss = 0.0
        eclass_running_acc = 0.0
        # reset evaluations for each 100 step
        cross_running_loss = 0.0
        class_running_acc = 0.0
        net.eval() # open eval mode
        for i, data in enumerate(val_data_loader):
            # put model and data to gpu if available
            inputs, labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = net(inputs) # forward pass
            # calculate cross-entropy loss
            loss = criterion(outputs, labels)
            cross_running_loss += loss.item()
            ecross_running_loss += loss.item()
            # classification accuracy
            outputs = F.softmax(outputs, dim=1) # softmax layer
            preds = torch.argmax(outputs, dim=1) # choose class with highest prob.
            results = torch.eq(labels, preds) # calculate ca
            # save the accuracy
            acc = torch.sum(results).item() / results.size(dim=0) # mean ca
            class_running_acc += acc
            eclass_running_acc += acc
            # update confusion matrix for validation
            update_confusion_matrix(val_confusion, preds, labels)

            if (i+1) % 100 == 0:
                print(f"VAL: [epoch {epoch + 1}, batch: {i + 1}] cross-entropy" +
                      f" loss: {round(cross_running_loss / 100, 3)}")
                print(f"VAL: [epoch {epoch + 1}, batch: {i + 1}] classification" +
                      f" accuracy: {round(class_running_acc / 100, 3)}")
                # save the results for each 100 steps
                val_cross_hist.append(cross_running_loss / 100)
                val_class_hist.append(class_running_acc / 100)
                cross_running_loss = 0.0
                class_running_acc = 0.0

        mean_acc = eclass_running_acc / len(val_data_loader)
        # save the results for each epoch
        eval_cross_hist.append(ecross_running_loss / len(val_data_loader))
        eval_class_hist.append(mean_acc)
        # save the best performance
        if mean_acc > max_acc_val:
            best_vconfusion = val_confusion
            max_acc_val = mean_acc

    # all evaluation metrics put into dictionaries for easy usage
    results_100step = {"train_cross_entropy" : train_cross_hist,
                       "train_class_acc" : train_class_hist,
                       "val_cross_entropy" : val_cross_hist,
                       "val_class_acc" : val_class_hist}

    results_epoch = {"train_cross_entropy" : etrain_cross_hist,
                     "train_class_acc" : etrain_class_hist,
                     "val_cross_entropy" : eval_cross_hist,
                     "val_class_acc" : eval_class_hist}

    confusion = {"train_confusion" : best_tconfusion,
                 "val_confusion" : best_vconfusion}
    print("training is completed!")

    return {"100steps" : results_100step,
            "epoch" : results_epoch,
            "confusion matrix" : confusion}

def plot_confusion_matrix(network_name, results, aug=False):
    title_aug = ""
    if aug:
        title_aug = "(Aug.)"
    # classes
    classes = ['airplane', 'bus', 'cat', 'dog', 'pizza']
    # plot the train confusion
    conf_matrix = results['confusion matrix']['train_confusion'].astype('i')
    acc = round(np.trace(conf_matrix) / np.sum(conf_matrix), 3)
    plt.figure()
    s = sns.heatmap(conf_matrix, annot=True, cmap='Blues', xticklabels=classes,
                    yticklabels=classes, fmt='g')

    s.set(xlabel='Predicted label', ylabel='True label')
    plt.title(f"{network_name}{title_aug} Train Accuracy=%{acc*100}")
    plt.show()
    # plot the validation confusion
    conf_matrix2 = results['confusion matrix']['val_confusion'].astype('i')
    acc2 = round(np.trace(conf_matrix2) / np.sum(conf_matrix2), 3)
    plt.figure()
    s2 = sns.heatmap(conf_matrix2, annot=True, cmap='Blues', xticklabels=classes,
                    yticklabels=classes, fmt='g')

    s2.set(xlabel='Predicted label', ylabel='True label')
    plt.title(f"{network_name}{title_aug} Validation Accuracy=%{acc2*100}")
    plt.show()

def plot_images(train_data):
    # class names
    classes = ["airplane", "bus", "cat", "dog", "pizza"]
    # plot random 5 images for each class
    fig, axs = plt.subplots(5, 5)
    fig.tight_layout()
    for i, cat in enumerate(classes):
        files = train_data[cat]
        chosen = random.sample(files, 5) # sample 5 random images
        for j, file_name in enumerate(chosen):
            img = Image.open(os.path.join('train_data', file_name)) # load image
            axs[i, j].imshow(img) # plot image
            axs[i, j].axis('off')
            axs[i, j].set_title(f"{cat} {j + 1}", size=7)
    plt.show()

def plot_evaluations(results):
    train_ce = [result['epoch']['train_cross_entropy'] for result in results]
    # plot train cross-entropy loss for epochs
    plt.title("Cross-entropy v.s. Epochs")
    plt.plot(train_ce[0])
    plt.plot(train_ce[1])
    plt.plot(train_ce[2])
    plt.xlabel("Epochs")
    plt.ylabel("Train Cross-entropy")
    plt.legend(["Net1", "Net2", "Net3"])
    plt.grid("ON", linewidth=0.5)
    plt.show()

    # plot classification accucarcy for epochs
    train_ca = [result['epoch']['train_class_acc'] for result in results]

    plt.title("Classification Accuracy v.s. Epochs")
    plt.plot(train_ca[0])
    plt.plot(train_ca[1])
    plt.plot(train_ca[2])
    plt.xlabel("Epochs")
    plt.ylabel("Train Classification Accuracy")
    plt.legend(["Net1", "Net2", "Net3"])
    plt.grid("ON", linewidth=0.5)
    plt.show()

    # plot cross-entropy ploss for each 100 iteration
    train_ce = [result['100steps']['train_cross_entropy'] for result in results]

    plt.title("Cross-entropy v.s. Steps")
    plt.plot(train_ce[0])
    plt.plot(train_ce[1])
    plt.plot(train_ce[2])
    plt.xlabel("Steps")
    plt.ylabel("Train Cross-entropy")
    plt.legend(["Net1", "Net2", "Net3"])
    plt.grid("ON", linewidth=0.5)
    plt.show()

    # plot classification accuracy for each 100 iteration
    train_ca = [result['100steps']['train_class_acc'] for result in results]

    plt.title("Classification Accuracy v.s. Steps")
    plt.plot(train_ca[0])
    plt.plot(train_ca[1])
    plt.plot(train_ca[2])
    plt.xlabel("Steps")
    plt.ylabel("Train Classification Accuracy")
    plt.legend(["Net1", "Net2", "Net3"])
    plt.grid("ON", linewidth=0.5)
    plt.show()


if __name__ == '__main__':
    # take the data directory as input
    parser = argparse.ArgumentParser()
    parser.add_argument("--data_dir", default="/Users/berksahin/Desktop", help="Current 
directory path of the coco folder")
    args = parser.parse_args()

    np.random.seed(5)
    random.seed(5)

    # prepare the dataset
    data = set_datasets(args.data_dir)
    plot_images(data["train_data"])
    # construct datasets
    train_dataset = MyCOCODataset(folder_name='train_data')
    val_dataset = MyCOCODataset(folder_name='val_data')
    # construct dataloaders
    train_loader = DataLoader(dataset=train_dataset, batch_size=4,
                              num_workers=2)
    val_loader = DataLoader(dataset=val_dataset, batch_size=4,
                            num_workers=2)

    # CNN Task 1
    model = HW4Net()
    result1 = train(model, train_loader, val_loader, epochs=15)

    # save the dictionary to the current directory
    with open('results1.pkl', 'wb') as f:
        pickle.dump(result1, f)
        print("results were saved succesfully to file results1.pkl")
    plot_confusion_matrix("Net 1", result1)

    # CNN Task 2
    model = Net2()
    result2 = train(model, train_loader, val_loader, epochs=15)

    # save the dictionary to the current directory
    with open('results2.pkl', 'wb') as f:
        pickle.dump(result2, f)
        print("results were saved succesfully to file results2.pkl")

    plot_confusion_matrix("Net 2", result2)

    # CNN Task 3
    model = Net3()
    result3 = train(model, train_loader, val_loader, epochs=15)

    # save the dictionary to the current directory
    with open('results3.pkl', 'wb') as f:
        pickle.dump(result3, f)
        print("results were saved succesfully to file results3.pkl")

    plot_confusion_matrix("Net 3", result3)

    # plot learning curves
    results = [result1, result2, result3]
    print("plots are generated!")
    plot_evaluations(results)
```