Deep Learning Assignment 2: Convolutional Neural Networks

Huey Ning Lok

November 14, 2019

1 Description:

In this assignment, we will implement a convolutional neural network in PyTorch. At a minimum, it should be solving a binary classification problem using a novel dataset. You should experiment with different architectures, activation functions, regularization techniques, optimizers, etc. and thoroughly explain which techniques you tried, which worked better, and why you made this specific architecture decision.

1.1 Requirements:

Implement a working convolutional neural network using the architecture of your choice. The architecture of your network should be thoroughly commented. Avoid using standard datasets such as MNIST or CIFAR10. The more creative you get the better! At a minimum, you should tackle binary classification problem. As an extension, you can try working on more complicated datasets, multiclass problems, etc.

Please document your code and write a brief summary which includes any interesting technical details and your results.

1.2 Extensions:

Implement one or more of the following, and include a comparison of the model's runtime and performance with/without each component:

- More than one architecture type: either experiment with different layers or implement a different architecture from the literature
- You can decide to tackle a more complicated problem such as object detection, segmentation, etc. In this case, you should justify the usage of a specific architecture.
- Train the network on more than 1 dataset.
- Compare your model to a pre-trained network.

2 Assignment Structure

For this assignment, I have decided to build and train a Convolutional Neural Network (CNN) model to solve a multilabel classification problem. I trained my custom model on the CIFAR dataset to ensure that it has a reasonable performance on a "classic" existing dataset before attempting to train my model on a novel, self-collected dataset. The custom model's performance on both the CIFAR and novel dataset are compared.

3 Building the Custom Model

In the PyTorch tutorial on training a classifier (2017), they used a basic CNN model as follows:

```
Input -> Conv -> Pool -> Conv -> Pool -> Linear -> Linear -> Linear (output)
```

The convolutional and linear layers were activated using the ReLU function.

Using this model as a base, I tweaked the architecture by heavily referencing Stanford's CS231n CNN course notes on how to design a CNN (Karpathy & Johnson, 2017). There were some design choices that, while recognized as having high potential to improve model performance, were omitted from the network due to time and resource constraints.

3.1 Layers and Operations

Convolutional Layer The convolutional layer is the main building block of the CNN. The convolutional layer performs a "convolution" by sliding a filter/kernel of randomly initialized weights over each pixel in the input image and taking the dot product of the filter weights and image pixel value at any given position. Once the filter has "convolved" the entire input, the result will be a 2D activation map of the filter responses at every spatial position. These activations can look totally incomprehensible in the first few iterations of training the network, when all the filter weights are completely random, but after awhile we expect to see some intuitive image features, e.g. edges, curves, being learned by the network through backpropagation and parameter updates (weights, biases).

Karpathy and Johnson (2017) advocated for **"a stack of small filter CONV to one large receptive field CONV layer"** on the basis that the combination of non-linear activations at each stack allowed the CNN to learn more expressive features, and that a stack of small-filter convolutional layers could achieve the same dimensional view of the input as a single large-filter convolutional layer while having less learnable parameters.

Pooling Layer The purpose of pooling layers are to "progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting" (Kaparthy & Johnson, 2017). Max pooling is most commonly used - a small window (typically with receptive field of 2x2 and stride of 2) slides across the input and takes the max number from each window, while discarding the rest. Since the window can be veiwed as a box with 4 numbers, this leads to about 75% of the input being discarded.

A pooling layer with receptive field, $F = 2$ and stride, $S = 2$ that accepts an input with dimensions of:

$W \times H \times D$

will produce an output with dimensions:

$\left(\frac{(W-F)}{S} + 1\right) \times \left(\frac{(H-F)}{S} + 1\right) \times D$

So, if the input is $(32, 32, 3)$, the output after max-pooling will be $(16, 16, 3)$.

Pooling layers helps to prevent overfit and reduces the amount of learnable parameters, but they can also be destructive since they discard away 75% of the input information.

Batch Normalization Batch normalization (BN) is a technique to normalize activations in intermediate layers of deep neural networks, it is normally performed over mini-batches and not the entire neural network due to speed reasons (Bjorck, Gomes, Selman & Weinberger, 2018).

At a high level, BN works by subtracting the mean activation from all input activations in a channel, then divides the centered activation by the standard deviation of the channel; this is followed by a channel-wise affine transformation parametrized by slope and intercept parameters which are learned during training (Bjorck et al., 2018).

BN has been hypothesized to reduce “internal covariate shift” – the tendency of the distribution of activations to drift during training, thus affecting the inputs to subsequent layers (Ioffe & Szegedy, 2015 as cited in Bjorck et al., 2018). While this hypothesis has not been proven to be infallibly true, empirical results have shown that networks with BN tends to result in higher accuracy vs non-BN networks (Bjorck et al., 2018).

Dropout Dropout is another technique to prevent overfitting. The idea behind it is to "randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much" (Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov, 2014). As I was already implementing max-pooling to reduce overfit, and a recent study showed that the combination of both dropout and batch normalization within a neural network often leads to worse performance unless using a very wide neural net (Li, Chen, Hu & Yang, 2019), I decided to omit a dropout operation from my neural network.

3.2 Activation Function

The Rectified Linear Unit (ReLU) has been shown to outperform the sigmoid activation function by greatly accelerating the convergence of stochastic gradient descent (probably due to its linear, non-saturating form, i.e. $f(x) = \max(0, x)$). However, ReLU units can also potentially "die" (remain at 0) when a very large gradient flows through the function and causes the weights to update in a way such that the neuron will never activate on any other datapoint it sees (Karpathy & Johnson, 2017).

Leaky ReLU are an attempt to fix the "dying ReLU" problem by allowing a small, positive gradient when the unit is not active (Maas, Hannun & Ng, 2013). That is:

$$f(x) = \begin{cases} x & x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Leaky ReLU does not always improve performance (Karpathy & Johnson, 2017), but I decided to use it in my neural network anyway since it solved the dying ReLU problem.

3.3 Optimizers

I initially wanted to test out an Adam optimizer since it was newer than SGD and supposedly designed for deep neural network training in particular (Kingma & Ba, 2014). However, after reading that it can often show worse results than a basic SGD optimizer over long training times (Bushaev, 2018), I decided to just use a SGD optimizer.

3.4 CNN Architectures

Karpathy and Johnson (2017) provided the following common CNN architectures:

```
INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC
```

This is similar to the PyTorch tutorial's architecture, i.e. each convolutional layer is followed by a pooling layer.

```
INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC
```

In this architecture, there are two convolutional networks before the pooling operation is performed, so that the network can "develop more complex features of the input volume before the destructive pooling operation" (Kaparthy & Johnson, 2017).

3.5 My CNN Architecture

I decided to follow the conventional structure suggested by Karpathy and Johnson (2017) of using `CONV -> RELU -> POOL` layers, but substituted the ReLUs with Leaky ReLUs instead.

Following the principle of using small kernel receptive fields, my convolutional layers had kernel sizes of 3 x 3. The kernels had a stride of 1 to reduce information loss, and no padding. While it was suggested to stack multiple convolutional layers, I only stacked two due to computational resources.

I performed max-pool operations after each convolution as a way to prevent overfit and extract the "most important" features from the spatial input.

I performed a 2D batch normalization after the second convolutional layer, and a 1D batch normalization after the first linear layer.

The last 3 layers of my model are linear layers. This was mostly going off Kaparthy and Johnson's (2017) advice that it was common to transition towards fully-connected linear layers

towards the end of the network, though I did not have a good intuition as to why this was so (except for the final output layer where the classification was performed).

The CNN takes in an input of size 32 x 32 x 3, and outputs an activation energy for each of the classes found in the dataset. For CIFAR, this would be 10 classes. For my novel dataset, this would be 3 classes.

The architecture of the model is shown below. A summary of the model with its outputs at every layer as well as number of parameters can be seen in the model training section.

```
In [1]: # import libraries
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns
        sns.set()

        # torch imports
        import torch
        import torchvision
        import torchvision.transforms as transforms
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        import torchvision.datasets
        import torchvision.utils as vutils

        # Number of GPUs available. Use 0 for CPU mode.
        ngpu = 1
        # Decide which device we want to run on
        device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

In [2]: # build the model
        class Net(nn.Module):

            '''
            Custom CNN Model.
            '''


            def __init__(self, out_features):

                super(Net, self).__init__()

                self.conv1 = nn.Conv2d(in_channels = 3,
                                       out_channels = 6,
                                       kernel_size = 3,
                                       stride = 1)

                self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)

                self.conv2 = nn.Conv2d(in_channels = 6,
                                       out_channels = 16,
                                       kernel_size = 3,
                                       stride = 1)

                self.conv2_bn = nn.BatchNorm2d(num_features = 16) # batch norm for conv layer

                self.fc1 = nn.Linear(in_features = 16 * 6 * 6,
                                     out_features = 120)

                self.dense1_bn = nn.BatchNorm1d(num_features = 120) # batch norm for linear layer

                self.fc2 = nn.Linear(in_features = 120,
```

```
                                out_features = 84)

        self.fc3 = nn.Linear(in_features = 84,
                                out_features = out_features)


    def forward(self, x):

        x = self.pool(F.leaky_relu(self.conv1(x)))
        x = self.pool(F.leaky_relu(self.conv2_bn(self.conv2(x))))
        x = x.view(-1, 16 * 6 * 6) # before linear layer, flatten output to 16*6*6
        x = F.leaky_relu(self.dense1_bn(self.fc1(x)))
        x = F.leaky_relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

4 Custom Model on CIFAR

The CIFAR dataset consists of 50,000 train images and 10,000 test images (train:test ratio of 5:1). It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size (32, 32, 3), i.e. 3-channel color images of (32, 32) pixels in size (Krizhevsky & Hinton, 2009, as cited in PyTorch, 2017).

4.1 Import and Preprocess Data

Since the PyTorch library already has a version of the CIFAR dataset, minimal preprocessing had to be done. The data was transformed to a tensor and the RGB color channels were normalized using a mean and standard deviation of 0.5 across all three channels. The batch size was set to be 32.

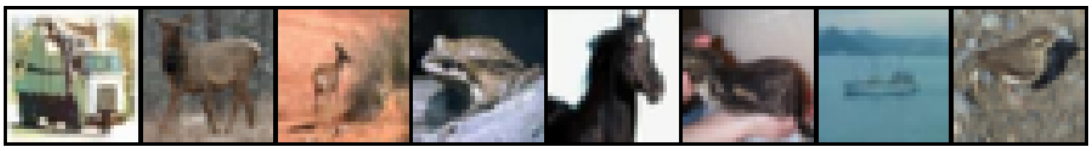
```
In [3]: # set batch size
        batch_size = 32

        # transform images to tensor and normalize
        transform = transforms.Compose(
            [transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

In [4]: # create dataloaders
        trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                download=True, transform=transform)
        trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                                  shuffle=True, num_workers=2)

        testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                               download=True, transform=transform)
        testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                                 shuffle=False, num_workers=2)
        classes = ('plane', 'car', 'bird', 'cat',
                   'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

Files already downloaded and verified
Files already downloaded and verified


In [5]: print(f"Number of training data points: {len(trainset)}")
        print(f"Number of testing data points: {len(testset)}")
```

```
Number of training data points: 50000
Number of testing data points: 10000
```

```
In [6]: def plot_images(dataloader, classes, image_number = 8, model = None):

            '''
            Function to plot a sample of images from the dataloader, alongside their class
        labels.
            If a model is assigned to the model parameter, the predicted labels will be printed
        as well.

            Input:
                dataloader (DATALOADER)
                    Dataloader of dataset.

                classes (ARR)
                    Array type object containing the class labels (strings) in the order that
                    corresponds with the numerical key in the dataloader.

                image_number (INT)
                    Number of images to plot from the dataloader. image_number should not exceed
        batch size.
                    Since images are plotted in a row, any number > 10 could cause display
        issues.
                    Default: 8.

                model (PYTORCH MODEL)
                    Optional parameter. If a model is provided, the predicted labels from the
        model for each of the
                    images will be printed as well.
                    Default: None.
            '''

            # get images and true labels
            images, labels = next(iter(dataloader))

            # plot images
            plt.figure(figsize=(16,16))
            plt.axis("off")
            plt.imshow(np.transpose(vutils.make_grid(images.to(device)[:image_number],
        padding=1, normalize=True).cpu(),(1,2,0)))

            # print true labels
            print('True labels: ', '     '.join('%5s' % classes[labels[j]] for j in
        range(image_number)))

            if model:
                # predict image classes using custom net
                outputs = model(images)
                # the outputs are energies for the 10 classes.
                # the higher the energy for a class, the more the network thinks that the image
        is of the particular class.
                # So, we get the index of the highest energy:
                _, predicted = torch.max(outputs, 1)
                # print predicted labels
                print('Predicted:  ', '   '.join('%5s' % classes[predicted[j]] for j in
        range(image_number)))
```

```
In [7]: # plot sample of training images
        plot_images(trainloader, classes)
```

```
True labels:  truck      deer      deer      frog      horse      cat      ship      bird
```



```
In [8]: from torchsummary import summary

        # create custom CNN model for CIFAR dataset
        cifar_net = Net(out_features = len(classes))
        print(cifar_net)
        summary(cifar_net, (3,32,32))
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2_bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (dense1_bn): BatchNorm1d(120, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 30, 30]             168
         MaxPool2d-2            [-1, 6, 15, 15]               0
            Conv2d-3           [-1, 16, 13, 13]             880
       BatchNorm2d-4           [-1, 16, 13, 13]              32
         MaxPool2d-5             [-1, 16, 6, 6]               0
            Linear-6                  [-1, 120]          69,240
       BatchNorm1d-7                  [-1, 120]             240
            Linear-8                   [-1, 84]          10,164
            Linear-9                   [-1, 10]             850
================================================================
Total params: 81,574
Trainable params: 81,574
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.10
Params size (MB): 0.31
Estimated Total Size (MB): 0.42
----------------------------------------------------------------
```

4.2 Model Training

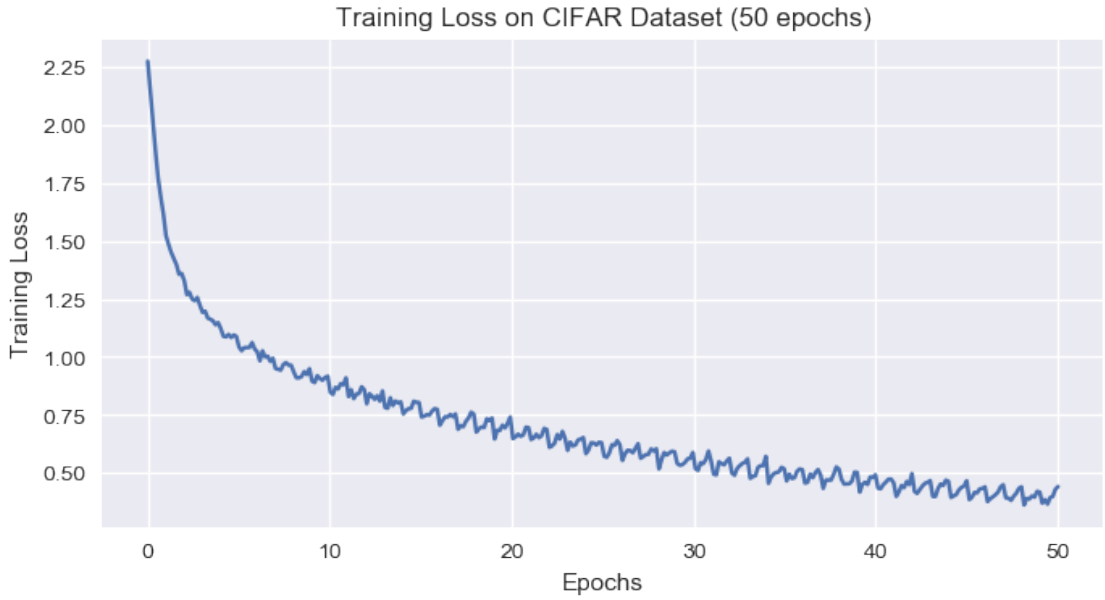
```
In [9]: def train_model(dataloader, model, criterion, optimizer, epochs = 10):
            '''
            Function to train a given model on a given dataset.

            Input:
                dataloader (DATALOADER)
                    Dataloader of dataset.

                model (PYTORCH MODEL)
                    PyTorch model to be trained.

                criterion (CRITERION)
                    Criterion (loss function) to be used in training.

                optimizer (OPTIMIZER)
                    Optimizer to be used in training.

                epochs (INT)
                    Number of training epochs.
                    Default: 10

            Output:
                train_loss (ARR)
                    Array of the model's running loss, calculated at every 200 iterations of training.

            '''

            train_loss = []

            for epoch in range(epochs):  # loop over the dataset multiple times

                running_loss = 0.0

                for i, data in enumerate(dataloader, 0): # loop through each batch

                    # get the inputs; data is a list of [inputs, labels]
                    inputs, labels = data

                    # zero the parameter gradients
                    optimizer.zero_grad()

                    # forward + backward + optimize
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    loss.backward()
                    optimizer.step()

                    # print statistics
                    running_loss += loss.item()
                    if i > 0 and i % 200 == 0:    # print running_loss at every 200 iterations
                        print(f"{epoch+1}, {i+1}, loss: {running_loss/200}")
                        train_loss.append(running_loss/200)
                        running_loss = 0.0

            print('Finished Training')
            return train_loss
```

```
In [10]: # set loss and optimizer
         criterion = nn.CrossEntropyLoss()
         optimizer = optim.SGD(cifar_net.parameters(), lr=0.001, momentum=0.9)

         train_loss = train_model(trainloader, model = cifar_net,
                                  criterion = criterion, optimizer = optimizer, epochs = 50)
```

```
1, 201, loss: 2.275298808813095
1, 401, loss: 2.1438225036859513
1, 601, loss: 2.006584076285362
.....................................
.....................................
50, 1001, loss: 0.3988352278620005
50, 1201, loss: 0.42947908222675324
50, 1401, loss: 0.4413864742219448
Finished Training
```

```
In [11]: # plot training loss
         plt.figure(figsize=(8,4), dpi=100)
         plt.title('Training Loss on CIFAR Dataset (50 epochs)')
         plt.xlabel('Epochs')
         plt.ylabel('Training Loss')
         plt.plot(np.linspace(0,50,len(train_loss)), train_loss)
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x1a233eb128>]
```



4.3 Model Testing

```
In [12]: # plot sample of testing images with predicted and true labels
         plot_images(testloader, classes, image_number=8, model=cifar_net)
```

```
True labels:    cat      ship      ship     plane      frog      frog       car       frog
Predicted:      cat     ship     ship    plane     frog     frog      car     frog
```

```
In [13]: def test_accuracy(dataloader, classes, model):
             '''
             Function for testing the accuracy of the model. The model weights are frozen when testing.

             Input:
                 dataloader (DATALOADER)
                     Dataloader of dataset to be tested on.

                 classes (ARR)
                     Array type object containing the class labels (strings) in the order that
                     corresponds with the numerical key in the dataloader.

                 model (PYTORCH MODEL)
                     Model to be used for testing.

             Output:
                 Prints the model's accuracy for each class label and the mean accuracy over all labels.

             '''

             class_correct = [0. for i in range(len(classes))]
             class_total = [0. for i in range(len(classes))]

             with torch.no_grad(): # set no grad for testing

                 for data in dataloader:
                     images, labels = data # get one batch of images and labels
                     outputs = model(images) # use model to predict labels from image batch
                     _, predicted = torch.max(outputs, 1) # get label index of class with highest energy
                     c = (predicted == labels).squeeze() # check whether predicted label == true label

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

             class_accuracy = [round(class_correct[i]/class_total[i]*100,2) for i in
         range(len(classes))]
             print(classes)
             print(class_accuracy)
             print(f"Mean Accuracy: {round(np.mean(class_accuracy),2)}")

In [14]: # compare accuracy on train vs test set
         print("Train Set Accuracy")
         test_accuracy(trainloader, classes, cifar_net)

         print("Test Set Accuracy")
         test_accuracy(testloader, classes, cifar_net)
```

```
Train Set Accuracy
('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
[88.81, 95.12, 85.44, 80.59, 82.08, 82.65, 87.15, 94.08, 93.23, 91.75]
Mean Accuracy: 88.09
Test Set Accuracy
('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
[74.0, 76.12, 54.55, 45.16, 48.53, 65.52, 67.86, 62.71, 74.6, 76.62]
Mean Accuracy: 64.57
```

5 Custom Model on Novel Dataset

The novel dataset consists of digital artworks scraped from artstation.com. The artworks can be seprated into three classes: "cyberpunk", "noir", and "cartoon". The original images are of size (400, 400, 3), i.e. 3-channel color images of (400, 400) pixels in size.

The original dataset is rather imbalanced; there are ~10,000 "cyberpunk" and "cartoon" images, but only ~4,000 "noir" images. With both time constraints and the imbalanced dataset as incentives, I decided to use only ~4,000 images from each class. The train:test ratio of the dataset was set to be 80:20.

5.1 Import and Preprocess Data

I resized the images from (400, 400, 3) to (32, 32, 3) to match the CIFAR dataset. This was an intentional choice for quicker computation, and also allowed me to use the same model structure for both datasets.

The batch size was set to 32, similar to the CIFAR training process. The RGB color channels were normalized using a mean and standard deviation of 0.5 across all three channels.

```
In [15]: # Use image folder dataset
         dataroot = 'images/'
         image_size = 32
         batch_size = 32

         transform = transforms.Compose([
                                        transforms.Resize(image_size),
                                        transforms.CenterCrop(image_size),
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                        ])
```

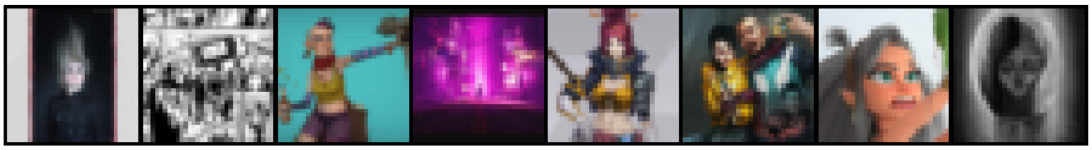
```
In [16]: def get_target_index(dataset):
             '''
             Given a dataset, this function returns a dictionary of classes, where the value of each class
             is a dictionary containing the class indices and the number of datapoints in the class.

             Input:
                 dataset (IMAGEFOLDER)
                     Dataset should be ImageFolder class.

             Output:
                 idx_dct (DCT)
                     Nested dictionary with the class name as key, and a dictionary containing the
                     'indices' and 'length' of the class as values.
                     Example format:
                     idx_dct = { 'class_A':{
                                 'indices': [1,2,3,4,5],
                                 'length': 5
                                 },
                                 'class_B':{
                                 'indices': [6,7,8],
                                 'length': 3
                                 },
                                 'class_C':{
                                 'indices': [100,101,102,103],
                                 'length': 4
                                 }}
             '''
             targets = torch.tensor([t[1] for t in dataset.samples])
```

```
        idx_dct = {}

        for k,v in dataset.class_to_idx.items():
            idx_dct[k] = {'indices': (targets == v).nonzero()}
            idx_dct[k]['length'] = len(idx_dct[k]['indices'])

        return idx_dct

    def create_train_test(dataset, batch_size, train_ratio = 0.8):
        '''
        Creates a train and test dataset by:
        1) balancing the dataset (shorten each class to the size of the min class length)
        2) slicing the class indices according to the train_ratio and test_ratio, i.e. 1 -
    train_ratio
        3) creating train and test samplers using the indices
        4) creating train and test loaders using the samplers

        Input:
            dataset (IMAGEFOLDER)
                Imagefolder type object containing the dataset.

            batch_size (INT)
                Batch size for creating the dataloader. This will impact the model training
                process as well.

            train_ratio (FLOAT)
                The ratio of training data to testing data. Value should be between 0 and 1.
                Default: 0.8

        Output:
            trainloader (DATALOADER)
                Dataloader of training set.

            testloader (DATALOADER)
                Dataloader of testing set.

        '''

        index_dct = get_target_index(dataset)

        # returns the length of the class with the min length
        balance_size = min(index_dct.items(), key=lambda x: x[1]['length'])[1]['length']
        train_size = int(train_ratio*balance_size)
        print(f"Each class is shortened to {balance_size} and split into training size of
    {train_size} and testing size of {balance_size-train_size}")

        train_indices = torch.cat([v[1]['indices'][:train_size].reshape(-1) for v in
    index_dct.items()], dim = 0)
        test_indices = torch.cat([v[1]['indices'][train_size:balance_size].reshape(-1) for v
    in index_dct.items()], dim = 0)

        print(f"Length of training indices: {len(train_indices)}")
        print(f"Length of testing indices: {len(test_indices)}")

        train_sampler = torch.utils.data.sampler.SubsetRandomSampler(train_indices)
        test_sampler = torch.utils.data.sampler.SubsetRandomSampler(test_indices)

        # create dataloaders
        trainloader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, sampler= train_sampler)
        testloader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, sampler =test_sampler)

        return trainloader, testloader

In [17]: # load dataset from my local directory
    dataset = torchvision.datasets.ImageFolder(root = dataroot, transform = transform)

    # create trainloader and testloader
    novel_trainloader, novel_testloader = create_train_test(dataset, batch_size, train_ratio= 0.8)
```

```
Each class is shortened to 3567 and split into training size of 2853 and testing size of 714
Length of training indices: 8559
Length of testing indices: 2142
```

```
In [18]: # plot sample of training images
         plot_images(novel_trainloader, dataset.classes, image_number=8)
```

```
True labels:   noir      noir      cartoon      cyberpunk      cyberpunk      cyberpunk      cartoon      noir
```



5.2 Model Training

```
In [19]: # create custom CNN model for CIFAR dataset
         novel_net = Net(out_features = len(dataset.classes))
         print(novel_net)
         summary(novel_net, (3,32,32))
```

```
Net(
  (conv1): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2_bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (dense1_bn): BatchNorm1d(120, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=3, bias=True)
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 6, 30, 30]             168
         MaxPool2d-2            [-1, 6, 15, 15]               0
            Conv2d-3           [-1, 16, 13, 13]             880
       BatchNorm2d-4           [-1, 16, 13, 13]              32
         MaxPool2d-5             [-1, 16, 6, 6]               0
            Linear-6                  [-1, 120]          69,240
       BatchNorm1d-7                  [-1, 120]             240
            Linear-8                   [-1, 84]          10,164
            Linear-9                    [-1, 3]             255
================================================================
Total params: 80,979
Trainable params: 80,979
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.10
Params size (MB): 0.31
Estimated Total Size (MB): 0.42
----------------------------------------------------------------
```

```
In [20]: # set loss and optimizer
         criterion = nn.CrossEntropyLoss()
         optimizer = optim.SGD(novel_net.parameters(), lr=0.001, momentum=0.9)

         # train the model
         novel_train_loss = train_model(novel_trainloader, model = novel_net,
                                        criterion = criterion, optimizer = optimizer, epochs =50)
```

```
1, 201, loss: 1.0734337297081948
2, 201, loss: 1.0117309030890465
3, 201, loss: 0.9854243528842926
.....................................................
.....................................................
48, 201, loss: 0.20643562380224467
49, 201, loss: 0.2001702824421227
50, 201, loss: 0.20168843919411303
Finished Training
```

```
In [25]: # plot training loss
         plt.figure(figsize=(8,4), dpi=100)
         plt.title('Training Loss on Novel Dataset (50 epochs)')
         plt.xlabel('Epochs')
         plt.ylabel('Training Loss')
         plt.plot(np.linspace(0,50,len(novel_train_loss)), novel_train_loss)
```

Out[25]: [<matplotlib.lines.Line2D at 0x1a24c679b0>]

5.3 Model Testing

```
In [22]: # plot sample of testing images with predicted and true labels
         plot_images(novel_testloader, dataset.classes, image_number=8, model=novel_net)
```

```
True labels:  cyberpunk    cartoon     noir    cartoon    cyberpunk    cyberpunk    cartoon    cyberpunk
Predicted:    cartoon    cartoon  cyberpunk  cyberpunk     noir    cyberpunk    cartoon     noir
```

```
In [23]: # compare accuracy on train vs test set
         print("Train Set Accuracy")
         test_accuracy(novel_trainloader, dataset.classes, novel_net)

         print("Test Set Accuracy")
         test_accuracy(novel_testloader, dataset.classes, novel_net)
```

```
Train Set Accuracy
['cartoon', 'cyberpunk', 'noir']
[92.59, 90.0, 95.81]
Mean Accuracy: 92.8
Test Set Accuracy
['cartoon', 'cyberpunk', 'noir']
[56.14, 60.61, 47.73]
Mean Accuracy: 54.83
```

6 Result Summary

6.1 CIFAR Performance

Train Set Accuracy

('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
[88.81, 95.12, 85.44, 80.59, 82.08, 82.65, 87.15, 94.08, 93.23, 91.75]
Mean Accuracy: 88.09

Test Set Accuracy

('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
[74.0, 76.12, 54.55, 45.16, 48.53, 65.52, 67.86, 62.71, 74.6, 76.62]
Mean Accuracy: 64.57

6.2 Novel Dataset Performance

Train Set Accuracy

['cartoon', 'cyberpunk', 'noir']
[92.59, 90.0, 95.81]
Mean Accuracy: 92.8

Test Set Accuracy

['cartoon', 'cyberpunk', 'noir']
[56.14, 60.61, 47.73]
Mean Accuracy: 54.83

6.3 Comparison

My custom model performs well during training on both datasets, reaching an accuracy of 88.09% on the CIFAR dataset, and an accuracy of 92.8% on my novel dataset. The mean test set accuracy for CIFAR is 64.57% (we expect an untrained model to have 10% accuracy since there are 10 classes), with the 'truck' class having the highest test accuracy of 76.62%, and 'cat' having the lowest test accuracy of 45.16%.

The mean test set accuracy for the novel dataset is 54.83% (we expect an untrained model to have a 33.33% accuracy since there are 3 classes), with the 'cyberpunk' class having the highest test accuracy of 60.61%, and 'noir' having the lowest test accuracy of 47.73%. Since I carefully balanced the dataset before splitting it into train and test sets, I do not think that this is due to dataset imbalance.

It should also be noted that while the training loss decreased exponentially when the model was training on CIFAR, it appears to decrease linearly when training on my novel dataset. This leads me to believe that the model may just be overfitting on the training set instead of actually "learning" how to classify the 3 classes. If the model was learning differentiable features, we expect an "elbow" point where the loss sharply decreases as the model has learned key features that can be used to classify new data. I am not sure what a linearly decreasing loss suggests, but it does not seem like the typical behavior of a model that is actually "learning" key features. My dataset was much smaller than CIFAR (train:test size of 8559:2142 vs CIFAR's 50,000:10,000), which could explain the overfit.

In the future, I would probably use more data for my model, augment the data to increase dataset size (e.g. through random horizontal flips), or experiment with different batch sizes since I used the same batch size of 32 for both models.

References

Bjorck, N., Gomes, C. P., Selman, B., & Weinberger, K. Q. (2018). Understanding batch normalization. In Advances in Neural Information Processing Systems (pp. 7694-7705).

Bushaev, V. (2018, October 24). Adam-latest trends in deep learning optimization. Retrieved November 14, 2019, from https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c.

Karpathy, A., & Johnson, J. (2017). CS231n: Convolutional Neural Networks for Visual Recognition-Transfer Learning. Retrieved November 13, 2019, from https://cs231n.github.io/transfer-learning/#tf.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images (Vol. 1, No. 4, p. 7). Technical report, University of Toronto.

Li, X., Chen, S., Hu, X., & Yang, J. (2019). Understanding the disharmony between dropout and batch normalization by variance shift. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2682-2690).

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June). Rectifier nonlinearities improve neural network acoustic models. In Proc. icml (Vol. 30, No. 1, p. 3).

PyTorch. (2017). Training a Classifier¶. Retrieved November 13, 2019, from https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

Ioffe, S. & Szegedy, C. (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning, pages 448–456.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

Appendix A: Pretrained Model Write-Up

I originally intended to compare my custom model's performance to a pretrained model, but was not able to due to time constraints. The paragraphs below show the literature research I had conducted in preparation of performing the transfer learning task.

The Pretrained Model

The pretrained model used was GoogLeNet, a 22 layers deep network that was trained on the ImageNet dataset (Szegedy, Liu, Jia, Sermanet, Reed, Anguelov,... & Rabinovich, 2015). I chose to use the pretrained model as a fixed feature extractor by freezing every layer of the network (except the final layer) and training a new final classification layer on the dataset. This is much more resource and time efficient compared to finetuning the model, i.e. continue training the weights via backpropogation on the new dataset.

According to Karpathy and Johnson (2017), when a dataset is large and similar to the original pretrained dataset (in this case, ImageNet), it is safer to finetune the pretrained model since there is lower probability of overfitting. When a dataset is small and very different from the original dataset, it is best to only train a linear classifier on the final layer to avoid overfitting. They also recommend training the classifier from "activations somewhere earlier in the network", since the top of the network might contain more features specific to the original data. However, they also mention that arbitrarily taking out Convolutional layers from the pretrained network may negatively affect performance.

CIFAR Dataset

Since the CIFAR dataset is large and similar to the ImageNet dataset, the pretrained model should work well as a fixed feature extractor. Finetuning the model could improve performance with minimal risk of overfit, but as mentioned above, the weights were frozen instead to save time and computational resource.

Novel Dataset

Since the dataset is both small and very different compared to the original ImageNet dataset, it made sense to use the pretrained GoogLeNet as a fixed-feature extractor.