CNN Model With PyTorch For Image Classification

Image size is 32X32

epochs number is 20

Data set is Herbarium Challenge 2019, small data set.
A convolutional neural network (CNN) takes an input image and classifies it into any of the output classes. Each image passes through a series of different layers – primarily convolutional layers, pooling layers, and fully connected layers.

The convolutional layer is used to extract features from the input image. It is a mathematical operation between the input image and the kernel (filter). The filter is passed through the image and the output is calculated.

PyTorch is one of the most popular and widely used deep learning libraries – especially within academic research. It's an open-source machine learning framework that accelerates the path from research prototyping to production deployment and I'll be using it in this practice to create my first CNN.

In [68]:
```
# Load in relevant libraries, and alias where appropriate
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torchvision import transforms
from torchvision.datasets import ImageFolder
import matplotlib.pyplot as plt
```

Transform Data

The torchvision.transforms module provides various functionality to preprocess the images, here first we resize the image for (30*30) shape and then transforms them into tensors(tensor is a way of representing the data in deep learning).

In [69]:
```
# Define relevant variables for the ML task
batch_size = 64
num_classes = 683
learning_rate = 0.001
num_epochs = 20

# Device will determine whether to run the training on GPU or CPU.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

To prepare a dataset from such a structure, PyTorch provides ImageFolder class which makes the task easy for us to prepare the dataset. I simply passed the directory of my data to it and it

provided the dataset which I could use to train the model.

The torchvision.transforms module provided various functionality to preprocess the images, here first I resized the image for (32*32) shape and then transforms them into tensors.

In [70]:
```
#train and test data directory
data_dir = "/Users/azadehbaghdadi/Desktop/New research-CNN/Project/Data/Herb/sma
test_data_dir = "/Users/azadehbaghdadi/Desktop/New research-CNN/Project/Data/Her


#load the train and test data
train_dataset = ImageFolder(data_dir,transform = transforms.Compose([
    transforms.Resize((32,32)),transforms.ToTensor()
]))
test_dataset = ImageFolder(test_data_dir,transforms.Compose([
    transforms.Resize((32,32)),transforms.ToTensor()
]))
```

So my first image in the dataset has a shape (3,32,32) which means the image has 3 channels (RGB), height 32, and width 32. The image has a label 0, which represents the “0” class.

The image label set according to the class index in test_dataset classes.

In [71]:
```
img, label = train_dataset[0]
print(img.shape,label)
```

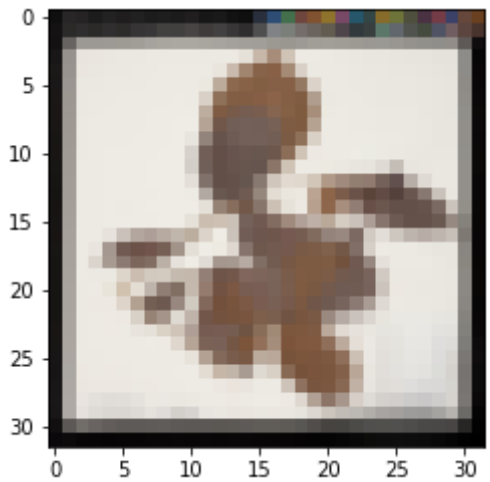
```
torch.Size([3, 32, 32]) 0
```

permute method reshapes the image from (3,32,32) to (32,32,3). The first image training data is of building as you can see below:

In [76]:
```
def display_img(img,label):
    print(f"Label : {train_dataset.classes[label]}")
    plt.imshow(img.permute(1,2,0))

#display the first image in the dataset
display_img(*train_dataset[0])
```

```
Label : 0
```



Splitting Data and Prepare Batches:

The batch size can be decided according to memory capacity, generally, it takes in power of 2. For example, the batch size can be 16, 32, 64, 128, 256, etc.

Here I took batches of size 128 and 2679 images from the data for validation and the 34225 of the data for training. The data set was divided previously in two training and valdation data set. Also I can randomly split the images into training and testing by PyTorch provides random_split(), that I did not do this here.

In [73]:
```
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split

batch_size = 128
val_size = len(test_dataset)
train_size = len(train_dataset) - val_size

train_data,val_data = random_split(train_dataset,[train_size,val_size])
print(f"Length of Train Data : {len(train_dataset)}")
print(f"Length of Validation Data : {len(val_data)}")
```

```
Length of Train Data : 34225
Length of Validation Data : 2679
```

The data is divided into batches using the PyTorch DataLoader class. I created two objects train_dl and val_dl for training and validation data respectively by giving parameters training data and batch size into the DataLoader Class.

In [74]:
```
#load the train and validation into batches.
train_dl = DataLoader(train_data, batch_size, shuffle = True, num_workers = 4, p
val_dl = DataLoader(val_data, batch_size*2, num_workers = 4, pin_memory = True)
```

Visualizing the images:

To visualize images of a single batch, make_grid() can be used from torchvision utilities. It gives us an overall view of images in batch in the form of an image grid.

In [75]:
```
from torchvision.utils import make_grid
import matplotlib.pyplot as plt

def show_batch(dl):
    """Plot images grid of single batch"""
    for images, labels in dl:
        fig,ax = plt.subplots(figsize = (16,12))
        ax.set_xticks([])
        ax.set_yticks([])
        ax.imshow(make_grid(images,nrow=16).permute(1,2,0))
        break

show_batch(train_dl)
```

CNN from Scratch

I started by define a new class that extends the nn.Module class from PyTorch. This is needed when you want to create a neural network as it provides you with a bunch of useful methods. I then defined the layers in my neural network. This is done in the **init** method of the class. I simply named my layers, and then assigned them to the appropriate layer that I wanted; e.g., convolutional layer, pooling layer, fully connected layer, etc. The final thing to do was define a forward method in my class. The purpose of this method is to define the order in which the input data passes through the various layers.

nn.Conv2d is used to define the convolutional layers. I defined the channels they receive and how much should they return along with the kernel size. I started from 3 channels, as my images are RGB images. nn.MaxPool2d is a max-pooling layer that just requires the kernel size and the stride, and nn.Linear is the fully connected layer, and nn.ReLU is the activation function used. In the forward method, I defined the sequence, and, before the fully connected layers, I reshape the output to match the input to a fully connected layer.

In [66]:

```
# Creating a CNN class
class ConvNeuralNet(nn.Module):
        #  Determine what layers and their order in CNN object
    def __init__(self, num_classes):
        super(ConvNeuralNet, self).__init__()
        self.conv_layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size
        self.conv_layer2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_siz
        self.max_pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.conv_layer3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_siz
        self.conv_layer4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_siz
        self.max_pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.fc1 = nn.Linear(1600, 128)
        self.relu1 = nn.ReLU()
```

```
        self.fc2 = nn.Linear(128, num_classes)

    # Progresses data across layers
    def forward(self, x):
        out = self.conv_layer1(x)
        out = self.conv_layer2(out)
        out = self.max_pool1(out)

        out = self.conv_layer3(out)
        out = self.conv_layer4(out)
        out = self.max_pool2(out)

        out = out.reshape(out.size(0), -1)

        out = self.fc1(out)
        out = self.relu1(out)
        out = self.fc2(out)
        return out
```

Setting Hyperparameters

I started by initializing my model with the number of classes. I then choosed cross-entropy and SGD (Stochastic Gradient Descent) as the loss function and optimizer respectively. There are different choices for these. I also define the variable total_step to make iteration through various batches easier.

In [67]:
```
model = ConvNeuralNet(num_classes)

# Set Loss function with criterion
criterion = nn.CrossEntropyLoss()

# Set optimizer with optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay =

total_step = len(train_dl)
```

Training

I started by iterating through the number of epochs, and then the batches in my training data, then I converted the images and the labels according to the device I used, i.e., GPU or CPU In the forward pass I made predictions using my model and calculated loss based on those predictions and omy actual labels. Next, I did the backward pass where I actually updated the weights to improve the model, then set the gradients to zero before every update using optimizer.zero_grad() function. Then, I calculated the new gradients using the loss.backward() function, and finally, I updated the weights with the optimizer.step() function.

As you can see, the loss is slightly decreasing with more and more epochs. This is a good sign.

In [56]:
```
# We use the pre-defined number of epochs to determine how many iterations to tr
for epoch in range(num_epochs):
        #Load in the data in batches using the train_loader object
```

```
    for i, (images, labels) in enumerate(train_dl):
        # Move tensors to the configured device
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

```
Epoch [1/20], Loss: 6.3719
Epoch [2/20], Loss: 6.0602
Epoch [3/20], Loss: 5.8639
Epoch [4/20], Loss: 6.1236
Epoch [5/20], Loss: 6.1440
Epoch [6/20], Loss: 6.0555
Epoch [7/20], Loss: 6.4170
Epoch [8/20], Loss: 6.2952
Epoch [9/20], Loss: 5.9227
Epoch [10/20], Loss: 5.8194
Epoch [11/20], Loss: 5.8662
Epoch [12/20], Loss: 5.9479
Epoch [13/20], Loss: 5.1748
Epoch [14/20], Loss: 5.8897
Epoch [15/20], Loss: 5.9646
Epoch [16/20], Loss: 5.9172
Epoch [17/20], Loss: 5.8291
Epoch [18/20], Loss: 5.8025
Epoch [19/20], Loss: 5.6628
Epoch [20/20], Loss: 5.5447
```

Testing

Let's now test my model. The code for testing is not so different from training, with the exception of calculating the gradients as we are not updating any weights:

In [57]:

```
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_dl:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the {} train images: {} %'.format(34225, 1
```

```
Accuracy of the network on the 34225 train images: 4.285804856400177 %
```

I wrap the code inside torch.no_grad() as there is no need to calculate any gradients. We then predict each batch using the model and calculate how many it predicts correctly. I got the final result of ~4% accuracy.