CNN Model With PyTorch For Image Classification

Image size is 256X256
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 this work, I started with 32x32 images and I got 4% acc in my first step. I increase the image size to 20X20 and the accuracy elevated to 20% which was not enough good yet. I am pretty sure this low accuracy was because of the low amount of data and a large number of classes. There were 34255 images and 683 classes.

In the second step, I decided to do data augmentation and increase the size of the data training set.

Data Augmentation:

flip horizontally and vertically

Image augmentation is one useful technique in building convolutional neural networks that can increase the size of the training set without acquiring new images. The idea is simple; duplicate images with some kind of variation so the model can learn from more examples. I use the OpenCV library to increase the number of my data training set images with flip horizontally and vertically and also rotating 90-degree methods.

My previous training set has almost 34000 images and after data Augmentation, it increased to more than 270000 images, almost 8 times.

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).

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.

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.

permute method reshapes the image from (3,32,32) to (32,32,3).

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.

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.

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.

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.

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.

Testing

Now I tested 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.

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 ~90% accuracy.

The final Result:

After doing data augmentation and also increasing the image size to 256X256 the accuracy increased to ~90% that it is pretty good.