A Quick Introduction to Computer Vision and Deep Learning with Neural Networks

Justin Dulay

As a freshly cropped researcher in computer vision, I want to maintain a blog to keep track of my progress while being able to also articulate it to more peers. So *please* provide feedback! Almost nothing that I post will be perfect, but I hope that it can help me grow as a researcher while also enhancing your reviewing abilities.

In this post, I want to discuss a notebook that I have been writing over the past few days to train a convolutional neural network on the MNIST dataset, while also employing my scratch Triplet Loss function.

First, I declare the imports that I will use.

```
import matplotlib.pyplot as plt
import numpy as np
import os
import random
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import datasets, models, transforms
```

Next, I set up this block to load in the images. I recoginize that this may also be done in a separate dataloaders file, and while that may be a more maintainable architecture longterm, the convenience of concise Jupyter notebooks is something new to me, after coming from a software development environment in a heavily compartmentalized collection of microservices.

So here, I first transform the images using PyTorch transforms. In order for the training data to fit the the ImageFolder train set, I needed to convert this to grayscale so that this tensor would exist on one channel instead of three.

Also, I wasn't sure exactly how to split the training data into training and validation sets, so this is *temporarily* utilizing the testing set for validation.

```
# images are 28x28
transformations = transforms.Compose([
    transforms.Resize(28),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
])

train_set = datasets.ImageFolder("/lab/vislab/DATA/MNIST/training/",
```

```
transform = transformations)
val_set = datasets.ImageFolder("/lab/vislab/DATA/MNIST/testing/",
transform = transformations)
print(train_set)

train_loader = torch.utils.data.DataLoader(train_set, batch_size=32,
shuffle=True)

val_loader = torch.utils.data.DataLoader(val_set, batch_size =32,
shuffle=True)

# torch.nn.TripletMarginLoss(margin=1.0, p=2.0, eps=1e-06, swap=False,
# size_average=None, reduce=None, reduction='mean')
# output = criterion(anchor, positive, negative)
criterion = MyTripletLoss()

# Connector to GPU
device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
print('Using device:', device)


# Sqaure size of each training image
img_size = 28

# Number of classes in the dataset
num_classes = 10

# Number of epochs to train for
num_epochs = 25
```

Here, I declare the model for my convolutional neural network. Again, I am open to suggestions on if I should change anything on this. For starters, I tried to matched the convolutional layers to link in output of the first to the input of the second. In the forward propogation, I also included a `RELU` hiddlen layer on each of the visible layers.

I certainly don't understand every single thing about this, so I will probably have more questions.

```
class Model_(nn.Module):
    """Basic model for this set. Any changes or suggestions are welcome"""

    def __init__(self):
        super(Model_, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6,
kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12,
kernel_size=5)

        self.fc1 = nn.Linear(in_features=12*4*4, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=60)
        self.out = nn.Linear(in_features=60, out_features=10)
```

```
    def forward(self, t):
        # conv 1
        t = self.conv1(t)
        t = F.relu(t)
        t = F.max_pool2d(t, kernel_size=2, stride=2)
        # conv 2
        t = self.conv2(t)
        t = F.relu(t)
        t = F.max_pool2d(t, kernel_size=2, stride=2)
        # fc1
        t = t.reshape(-1, 12*4*4)
        t = self.fc1(t)
        t = F.relu(t)
        # fc2
        t = self.fc2(t)
        t = F.relu(t)
        # output
        t = self.out(t)
        # don't need softmax here since we'll use cross-entropy as
activation.
        return t
```

Here, I define my custom Triplet Loss function. I know that PyTorch already has a `TripletMarginLoss` utility included, but I had considerable trouble setting it up with my dataset and its dimensions. Also, this loss function is *very* slow when run, but I am not sure exactly how to speed it up yet, which I will approach over next week.

For reference, here is the equation for triplet loss: $$L(A,P,N)=max(||f(A)- f(P)||^2 - ||f(A)- f(N)||^2 + \alpha, 0) $$

```
class MyTripletLoss(nn.Module):
    def __init__(self):
        super(MyTripletLoss, self).__init__()

    def forward(self, inputs, labels):
        # so inputs and labels are matrices
        losses = []
        batch_loss = 0.0
        # assume inputs and labels are same length

        for idx, anchor in enumerate(inputs):
            positive = random.choice([image_ for i, image_ in
enumerate(inputs) if labels[i] == labels[idx]])
            negative = random.choice([image_ for i, image_ in
enumerate(inputs) if labels[i] != labels[idx]])

            # safety of deep copy
            a1 = anchor.clone().detach()
            a2 = anchor.clone().detach()

            dist1 = a1.sub(positive)
```

```
            dist2 = a2.sub(negative)
            dist1 = dist1**2
            dist2 = dist2**2
            loss = max(dist1 - dist2 + 0.01)

            losses.append(loss)

        batch_loss = max(losses)
        return batch_loss
```

Here, I attempt to train an evaluate the model. Under each set epoch, the loss function is applied to batches to determine gradient descent between variables within each tensor.

```
def train_model(model, optimizer, num_epochs):
    """Bodied function to train data to network"""

    train_loss = 0.0
    loss = 0.0
    # sampler = torch.utils.data.RandomSampler(train_set,
replacement=False, num_samples=1)
    # eval for train and test, use criterion and back propagation,
specific towards
    # the type of loss that we want, contrastive
    for epoch in range(num_epochs):
        print("Epoch num: ", epoch)
        model.train()
        for idx, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            output = model.forward(inputs)

            loss = criterion(output, labels)
            loss = Variable(loss, requires_grad = True)

            loss.backward()

            optimizer.step()

            train_loss += loss.item()*inputs.size(0)

        print("loss for the epoch is:", train_loss)
        model.eval()
        val_loss = 0
        accuracy = 0
        counter = 0
        with torch.no_grad():

            for inputs, labels in val_loader:
                # Move to device
```

```
                inputs, labels = inputs.to(device), labels.to(device)
                # Forward pass
                output = model.forward(inputs)
                # Calculate Loss

                valloss = criterion(output, labels)
                valloss = Variable(valloss, requires_grad = True)



                # ******* THIS PART WAS FOUND ONLINE *********
                #
                #
                # Add loss to the validation set's running loss
                val_loss += valloss.item()*inputs.size(0)

                # Since our model outputs a LogSoftmax, find the real
                # percentages by reversing the log function
                output = torch.exp(output)
                # Get the top class of the output
                top_p, top_class = output.topk(1, dim=1)
                # See how many of the classes were correct?
                equals = top_class == labels.view(*top_class.shape)
                # Calculate the mean (get the accuracy for this batch)
                # and add it to the running accuracy for this epoch
                accuracy +=
torch.mean(equals.type(torch.FloatTensor)).item()

                # Print the progress of our evaluation
                counter += 1
                print(counter, "/", len(val_loader))
                #
                #
                #
                # ***********************************************

    return model, total_loss
```

In the block above, I will admit that the evaluation part was mostly sorted by an extenal resource. I want to prevent this as much as I can and originally write as much as possible from an original standpoint.

I currently have not been able to set up some plots, so I plan on improving with this after I ensure the network works.

I am completely open to more feedback in this, too! I am not sure if this can be an every week thing, but I will try to do so if possible.