Some resources for PRNN assignments

Deep Patel

April 1, 2020

I am putting up a few links and code snippets to help you with base-level implementation of SVMs and Neural Networks (NNs) along with some tid-bits on data-handling. Most of this stuff is, I suppose, for absolute beginners but do skim through it once. I have tried to keep the amount of text to a minimum in order to give you a quick summary. In the process, I may have ended up creating some voids in terms of clarity. Any form of feedback would be much appreciated.

Note #1: All the content has been taken from **scipy lectures** and official websites of the packages used (SciPy, Keras, Tensorflow, and PyTorch). I must add that the documentation provided by the scikit-learn community (in fact, the whole `Python` community in general) is quite comprehensive and most importantly, accessible, and hence don't hesistate to consult the beautifully-crafted documentation and examples. If you have any doubts, please reach out to `deeppatel@iisc.ac.in` or `santhoshg@iisc.ac.in`. Don't hesitate!

Note #2: As of now, I have created this document keeping in mind the minimal set of things required for your assignments. If you think there's something that's missing or that you would like for it to be added here, please let me know via email (`deeppatel@iisc.ac.in`).

Data-handling

The data provided in the assignments consists mainly of `.txt`, `.csv` or `.mat` files. The following links should help you load the data in Python:

- For `.txt` files: numpy_load_txt
- For `.csv` files: quick_intro_pandas, read_from_csv_pandas
- For `.mat` files: scipy_load_mat

In addition to this, the following document quite succintly summarizes the features of NumPy: guide_to_numpy

ML Algorithms

1. **Support Vector Machines (SVMs):**

   (a) **Snippet #1**

```
from sklearn import model_selection, datasets, metrics
from sklearn.svm import SVC

digits = datasets.load_digits()

#data split
X = digits.data
y = digits.target
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X,
                y, test_size=0.25, random_state=0)

#Train SVM
clf = SVC(gamma='auto').fit(X_train, y_train)
y_pred = clf.predict(X_test)
print('%s : %s ' % (Model.__name__, metrics.f1_score(y_test,
                y_pred, average="macro")))

print('-----------------')

#Test predictions
y_pred = clf.predict(X_test)
```

(b) **Snippet #2**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.datasets import make_blobs

# we create 40 separable points
X, y = make_blobs(n_samples=40, centers=2, random_state=6)

# fit the model, don't regularize for illustration purposes
clf = svm.SVC(kernel='linear', C=1000)
clf.fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

# plot decision boundary and margins
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
                   linestyles=['--', '-', '--'])

# plot support vectors
```

```
ax.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
           s=100, linewidth=1, facecolors='none', edgecolors='k')
plt.show()
```

(c) **Snippet #3**

```
#custom kernel
def my_kernel(X, Y):
        """
        We create a custom kernel:
                  (2   0)
        k(X, Y) = X  (       ) Y.T
                        (0   1)
        """
    M = np.array([[2, 0], [0, 1.0]])
    return np.dot(np.dot(X, M), Y.T)

h = .02  # step size in the mesh

# we create an instance of SVM and fit our data.
clf = svm.SVC(kernel=my_kernel)
clf.fit(X, Y)
```

2. **Multi-class classification in scikit-learn:**

   (a) The following webpage contains snippets for multi-class classification: https://scikit-learn.org/stable/modules/multiclass.html#multiclass

3. **Neural Networks:** So, typically, we use Keras/Tensorflow/PyTorch for training neural networks these days. There's a whole zoo of packages available. The three I have mentioned just now are the ones that I have used in one form or the other and hence I will be sharing code snippets only from these frameworks. You are free to choose and use your favourite package, however, the aforementioned three packages have the largest support in the ML research community and it would be advisable to choose from them.

   (a) **Keras:** It's a powerful API with built-in functions and more for the most commonly used neural networks and hence it's quite user-friendly. There's a good quick-intro available **here**, **here**, and **here**. The code provided below is an example of a CNN trained on CIFAR-10 dataset. It will give you a good idea of a lot of features of Keras as well as the typical pipeline for training a neural network. **And lastly, if someone wants to use Tensorflow, then all these codes will work with the only exception being `from tensorflow import keras` instead of `import keras`.** This code has been taken from **here**.

```
#Keras/Tensorflow API

from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import os

# specify the hyper-parameters and file-handling info
batch_size = 32
num_classes = 10
epochs = 100
data_augmentation = True
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'keras_cifar10_trained_model.h5'


# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')


# Convert class vectors to one-hot vectors
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# Define your neural network layers here
# The layers are defined with the help of
# keras.layers
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
```

```
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))


# initiate RMSprop optimizer
opt = keras.optimizers.RMSprop(learning_rate=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

# Normalize the data
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# Since CNN's are only translation invariant,
# we need to do a bit of pre-processing for
# the image dataset that we will be using
if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
    featurewise_center=False,  # set input mean to 0 over the dataset
    samplewise_center=False,  # set each sample mean to 0
        # divide inputs by std of the dataset
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,  # divide each input by its std
    zca_whitening=False,  # apply ZCA whitening
    zca_epsilon=1e-06,  # epsilon for ZCA whitening
```

```
        # randomly rotate images in the range (degrees, 0 to 180)
    rotation_range=0,
    # randomly shift images horizontally (fraction of total width)
    width_shift_range=0.1,
    # randomly shift images vertically (fraction of total height)
    height_shift_range=0.1,
    shear_range=0.,  # set range for random shear
    zoom_range=0.,  # set range for random zoom
    channel_shift_range=0.,  # set range for random channel shifts
    # set mode for filling points outside the input boundaries
    fill_mode='nearest',
    cval=0.,  # value used for fill_mode = "constant"
    horizontal_flip=True,  # randomly flip images
    vertical_flip=False,  # randomly flip images
    # set rescaling factor (applied before any other transformation)
    rescale=None,
    # set function that will be applied on each input
    preprocessing_function=None,
    # image data format, either "channels_first" or "channels_last"
    data_format=None,
    #fraction of images reserved for validation(strictly between 0 and 1
    validation_split=0.0)

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)


    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        epochs=epochs,
                        validation_data=(x_test, y_test),
                        workers=4)


# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)

model_path = os.path.join(save_dir, model_name)
model.save(model_path)
print('Saved trained model at %s ' % model_path)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
```

```
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

i. There's one more thing that I would like to add and that is the TENSORBOARD callback available in Keras/Tensorflow API. I would urge you to refer to the following link and see it for yourself: `tensorboard` and `tensorboard`.This tool will help you visualize the training process of a neural network in a much better way.

(b) **PyTorch:** You will be using `torch.nn` module to define your neural network layers here. (This is similar to the way we have used `keras.layers` above.) However, there's more to this module as it contains various other modules for parameter updation, loss function specification, etc. For a better understanding of the `torch.nn` module, please refer to this great `Jupyter notebook`.

```
# Define the neural network

import torch import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels,
        # 3x3 square convolution

        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        # 6*6 from image dimension
        self.fc1 = nn.Linear(16 * 6 * 6, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only
        # specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x

    def num_flat_features(self, x):
        # all dimensions except the batch dimension
        size = x.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
print(net)

# The learnable parameters of a model are returned by net.parameters()

params = list(net.parameters())
print(len(params))
print(params[0].size())  # conv1's .weight

# Zero the gradient buffers of all parameters
# and backprops with random gradients:

net.zero_grad()
out.backward(torch.randn(1, 10))

# define loss function
output = net(input)
target = torch.randn(10) # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)

#backprop
net.zero_grad()  # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
## parameter updation the naive way
learning_rate = 0.01 for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)

## parameter updation the right way

import torch.optim as optim
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update

##########################################
##########################################

# Now off to an actual ML application:

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

#Define the CNN
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
```

```
        return x

# train on GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assuming that we are on a CUDA machine,
#this should print a CUDA device:
print(device)


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          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=4,
                                         shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse',
'ship', 'truck')

# Network config
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Train the network
for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # print statistics
```

```
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')

# Save the model

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

## Testing the network

# First, load the trained model
net = Net()
net.load_state_dict(torch.load(PATH))

# predictions
outputs = net(images)

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%'
      % (100 * correct / total))
```

i. Note that `torch.nn` only supports mini-batches. The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and not a single sample.
   A. For example, `nn.Conv2d` (a convolution layer) will take in a $4-D$ tensor (essentially a multi-dimensional array) of size $n_{Samples} \times n_{Channels} \times Height \times Width$.
   B. If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

ii. A lot of bugs in PyTorch codes have been there because of having forgotten to put in the following *elusive* statement : `optimizer.zero_grad()`. It's such a recurring problem and annoyance that there are memes dedicated to this *phenomenon* on the internet. So, just giving you a heads up. :)

iii. Setting up `tensorboard` for PyTorch: https://pytorch.org/tutorials/intermediate/tensorboard_

Oh you read till the end? Great! In that case, I must say one last thing: Presently, PyTorch and Tensorflow, as you know, are the two most popular packages out there. PyTorch is **way way more Pythonic** than Tensorflow. Tensorflow does have something called *Eager Execution* (and it's improved a lot since Tensorflow 2.0 has come out) but the thing is it's still a headache at times to implement things like control flow statements (if-else, while, for loops) among other things. PyTorch is so user-friendly owing to a *dynamic computation graph* philosophy. This will be a huge digression but if you guys are interested (let me know via email), I can compile a detailed documentation about the differences between PyTorch and Tensorflow via discussions on computation graphs and so on but till then know this, try out both Keras/Tensorflow and PyTorch and pick your favourite package. There are things to be gained *and* lost by using either PyTorch or Tensorflow. So, do try out both though unlike me who is reluctant to try out PyTorch after *years* of using Keras/Tensorflow.