PyTorch: Completely Uninformed Image Modelling

Richie Morrisroe

September 12, 2017

Intro

- PyTorch is a framework for GPU computation in Python
- Re-implementation of Torch, which was a project in Lua

How do I install it?

```
conda env create pytorch
source activate pytorch
conda install pytorch torchvision -c soumith
```

- This is for CUDA 7.5 (which must be installed), Python 3 and does require Conda
- If you use Python, you should already be using conda
- Note that the above is taking a while
- Like long enough that's annoying me as I write this
- Reproducibility is a harsh mistress :/

Getting Started

```
import torch as torch
import torchvision as tv
```

- torch is the main package
- Loads of subpackages (autograd, nn and more)
- torchvision is a collection of uilities for image learning
  - it has transforms, and a dataset and model library

Basic Torch

```
x = torch.Tensor(5, 3)
y = torch.rand(5, 3)
x + y
```

Overall API

- torch: Tensor (i.e. ndarray) library
- torch.autograd: automatic differentiation
- torch.nn: a neural network library
- torch.optim: standard optimisation methods (SGD, RMSProp, etc)
- torch.multiprocessing: magical memory sharing
- torch.utils: loading and training utility functions
- Check out their about page

Sizes and Shapes

```
print(y.size())
```

Addition Ops

```
x + y
z = torch.zeros([5, 3])
th.add(x, y)
print(th.add(x, y))
th.add(z, y, out=z)
x.add_(y)
```

- Addition can be done in multiple ways
- We can assign the value to an out tensor as in the last example
- functions prepended with an _ mutate their first argument
- This is pervasive throughout the library

Indexing

```
print(x[:,1])
#x[R,C]
```

- Standard row, column indexing
- Note that 3d Tensors (i.e. for images) can have a minibatch dimension giving the number of observations in each minibatch
- Numpy uses H * W * C, Torch uses C * H * W
- This requires conversions to transpose the matrices (examples below)

Numpy Interaction

```
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
```

CUDA (yay!)

```
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    z = x + y

z_cpu = z.cpu()
```

- CUDA interop is easy
- Converting back is a simple cpu call

Autograd

```
from torch.autograd import Variable
x = Variable(th.ones(2, 2), requires_grad=True)
y = x + 2
print(y)
z = y * y * 3
out = z.mean()
print(z, out)
out.backward(retain_variables=True)
```

- This is probably the coolest thing about PyTorch
- Implements full reverse mode auto-differentiation
- This is done efficiently with a combination of memoizing and recursive applications of the chain rule
- These variables are inherently stateful, and thus idempotency is not preserved
- So eventually, repeated calls to backward leave you with a constant

Gradients

```
x = torch.randn(3)
x = Variable(x, requires_grad=True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2

print(y)
gradients = torch.FloatTensor([0.1, 1.0, 0.0001])
y.backward(gradients)
print(x.grad)
```

Transrforms

```
import torch
import torchvision
import torchvision.transforms as transforms
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5),
                          (0.5, 0.5, 0.5))]

)
```

- Transforms are applied at load time
- `RandomCrop` can be used, which is data augmentation

DataLoaders

```
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(trainset,
                                         batch_size=4,
                                         shuffle=True,
```

More datasets

- Some datasets are built in
- cifar10, 100, MNIST and a few others can be downloaded in this way
- Not ImageNet, which is only freely available to academics
- we can shuffle, alter the batch size and launch multiple processes easily
- transforms and image augmentation methods are additionally available
- This is actually in a package called `torchvision`, which has (as the same suggests) lots of utility functions related to vision

Adding your own data (easy way)

- I mostly copied from the transfer learning tutorial
- The approach relies on putting your data into specific folders

```
ls -R new_photos
```

- so the pattern is /data_dir/train/class/images
- you can then use the `datasets.ImageFolder` dataloader
- so we need a `train` and `val` folder
- We then need folders for each class (in this case, low, medium and high)

Loading Data

```
data_dir = 'new_photos'
dsets = { x: datasets.ImageFolder(os.path.join(data_dir, x),

dset_loaders = {x: torch.utils.data.DataLoader(dsets[x],
                                               batch_size=6,
                                               shuffle=True,
                                               num_workers=4
                                   for x in ['train', 'val']}
dset_sizes = {x: len(dsets[x]) for x in ['train', 'val']}
dset_classes = dsets['train'].classes
```

Dataloaders

- When I first looked at the code above, I was horrified. It seemed far too complicated for what it did.
- I replaced it with this:

```
from scipy import misc
test = misc.imread("new_photos/train/high/6813074_....jpg")
```

- However, the first solves way more problems
  - It implements lazy-loading which is good because each image is reasonably large
  - it shuffles the data
  - It varies the batch size (which can make a big difference)

Better dataloading

- Torch provides a `DataSet` class
  - Implement `__len__` and `__getitem__`

DataLoading

```
from skimage import io, transform
from torch.utils.data import Dataset, DataLoader
class RentalsDataSetOriginal(Dataset):
    def __init__(self, csv_file, image_path, transform):
        self.data_file = pd.read_csv(csv_file)
        self.image_dir = image_path
        if transform:
            self.transform = transform

    def __len__(self):
        return len(self.data_file)
```

Getitem

```
def __getitem__(self, idx):
    row = self.data_file.iloc[idx,:]
    dclass, listing, im, split = row
    image = io.imread(os.path.join(self.image_dir,
                                   split,
                                   dclass,
                                   im)).astype('float')
    img_torch = torch.from_numpy(image)
    h, w, c = img_torch.size()
    img_rs = img_torch.view([c, h, w])
    return (img_rs, dclass)
```

Features of Nets

- You must implement an init method
- This has the structure of the Net
- You must implement a forward method
- This should consist of all of non-linearites applied to each of the input layers
- PyTorch handles all the backward differentiations for you

Minimal Neural Network

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 48, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(48, 64, 5)
        self.conv3 = nn.Dropout2d()
        #honestly, I just made up these numbers
        self.fc1 = nn.Linear(64*29*29, 300)
        self.fc2 = nn.Linear(300, 120)
        self.fc3 = nn.Linear(120,3)
```

- the `__init__` method creates the structure of the net
- You need to provide input and output sizes
- If you mess this up, comment out all of the layers after the error, and use `x.size()` to decide what to do
- must inherit from nn.Module (or a more specific version)

Forward Differentiation

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 64 * 29 * 29) #-1 ignores the minibat
    x = F.dropout(x, training=self.training)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

- The forward operator contains the non-linearities
- Note the training argument to dropout

Training the Model

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimiser = optim.SGD(net.parameters(), lr=0.01,
                      momentum=0.9)
tr = dset_loaders['train']
for epoch in range(10):
    for i, data in enumerate(tr, 0):
        inputs, labels = data
        inputs, labels = Variable(inputs.cuda()),
        Variable(labels.cuda())
        optimiser.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        _, preds = torch.max(outputs.data, 1)
        loss.backward()
        optimiser.step()
```

Saving Model State

```
dtime = str(datetime.datetime.now())
    outfilename = 'train' + "_"
    + str(epoch) +  "_"
    + dtime + ".tar"
    torch.save(net.state_dict(), outfilename)
```

- Useful to resume training
- Current model state can be restored into a net of exactly the same shape
- Not as important for my smaller models
- These files are huuuuuggggeeee
- So you may wish to only save whichever performs best

Testing the Model

```
for epoch in range(5):
    val_loss = 0.0
    val_corrects = 0
    for i, data in enumerate(val, 0):
        inputs, labels = data
        inputs, labels = Variable(inputs.cuda()),
        Variable(labels.cuda())
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        _, preds = torch.max(outputs.data, 1)
        val_loss += loss.data[0]
        val_corrects += torch.sum(preds == labels.data)
        phase = 'val'
    val_epoch_loss = val_loss / dset_sizes['val']
    val_epoch_acc = val_corrects / dset_sizes['val']
    print('{} Loss: {:.4f}  Acc: {:.4f}'.format(
            phase, val_epoch_loss, val_epoch_acc))
```

Playing with the Net

```
params = list(net.parameters())
print(len(params))
print(params[0].size())

input = Variable(torch.randn(3, 3, 48, 48))
out = net(input)
print(out)
```

How did it do?

```
train Loss: 0.1360 Acc: 0.6742
train Loss: 0.1355 Acc: 0.6750
...
train Loss: 0.1202 Acc: 0.6966
val Loss: 0.1432  Acc: 0.6816
...
val Loss: 0.1440  Acc: 0.6810
```

- Training Accuracy 69% (10 epochs)
- Test Accuracy 68%
- This is OK, but given the data and the lack of any meaningful domain knowledge, I'm reasonably impressed.
- I guess what we actually need to know is what the incremental value of the image data is, relative to the rest of the data.

Text Data

- Fortunately, the rentals dataset also has some text data

```
import pandas as pd
text = pd.read_csv("rentals_sample_text_only.csv")
first = text.iloc[0,:]
print(list(first))
```

= »> »> ['This location is one of the most sought after areas in Manhattan ** Building is located on an amazing quiet tree lined block located just steps from transportation, restaurants, boutique shops, grocery stores*** For more info on this unit and/or others like it please contact Bryan 449-593-7152 / kagglemanager@renthop.com

Bond New York is a real estate broker that supports equal housing opportunity.<p><a website_redacted '] =in Manhattan Building is located on an amazing quiet tree lined block located just steps from transportation, restaurants, boutique shops, grocery stores*** For more info on this unit and/or others like it please contact Bryan 449-593-7152 / kagglemanager@renthop.com

 Bond

Characters vs Words?

- Most NLP that I traditionally saw used words (and bigrams, trigrams etc) as the unit of observation
- Many deep learning approaches instead rely on characters
- Characters are much less sparse than words
- We have way more characters
- We don't understand a word as a function of its characters, so should a machine?

Characters

- They are much less sparse
- The representation is pretty cool also
- We represent each character as a 1*N tensor for each item in the character universe
- Each word is represented as a matrix of these characters

Preprocessing

```
import unicodedata
import string

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

def unicode_to_ascii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )
```

- Cultural Imperialism rocks!
- More seriously, we reduce the dimension from 90+ to 32
- This means we can handle more words and longer descriptions

Apply to the text data

```
first = text['description']
first2 = []
char_ascii = {}
for word in first:
    for char in word:
        char = unicode_to_ascii(char.lower())
        if char not in char_ascii:
            char_ascii[char] = 1
        else:
            pass
```

- We need the character counts to create a mapping from characters to a 1-hot matrix
- This is necessitated by the disappointing lack of R's `model.matrix`
- This code was also used to assess the impact of removing non-ascii chars

Character to Index

```
import torch
all_letters = char_ascii.keys()
letter_idx ={}
for letter in all_letters:
    if letter not in letter_idx:
        letter_idx[letter] = len(letter_idx)


def letter_to_index(letter):
    return letter_idx[letter]
```

- Create a dict with the key being the number of previous letters
- Use this to represent the letter as a number

Letter/Words to Tensor

```
def letter_to_tensor(letter):
    tensor = torch.zeros(1, len(char_ascii))
    tensor[0][letter_to_index(letter)] = 1
    return tensor

def line_to_tensor(line):
    tensor = torch.zeros(len(line), 1, len(char_ascii))
    for li, letter in enumerate(line):
        letter = unicode_to_ascii(letter.lower())
        tensor[li][0][letter_to_index(letter)] = 1
    return tensor
```

- Code implementation for the character and word to tensor functions
- Note that these are going to be really sparse vectors (1 non-sparse entry per row)
- torch has sparse matrix support (but it's marked as experimental)

Bespoke Rentals Code

```
all_categories = ['low', 'medium', 'high']
def category_from_output(output):
    top_n, top_i = output.data.topk(1)
    category_i = top_i[0][0]
    return all_categories[category_i], category_i
```

- We need to be able to map back from a matrix of probabilities to a class prediction

Different Get Data Implementation

```
import pandas as pd
textdf = pd.read_csv('rentals_text_data.csv').dropna(axis=0)
cat_to_ix = {}
for cat in all_categories:
    if cat not in cat_to_ix:
        cat_to_ix[cat] = len(cat_to_ix)
    else:
        pass

def random_row(df):
    rowrange = df.shape[0] - 1
    return df.iloc[random.randint(0, rowrange)]
```

Shuffling Training Examples

```
import random as random
from torch.autograd import Variable
def random_training_example(df):
    row = random_row(df)
    target = row['interest_level']
    text = row['description']
    catlen = len(all_categories)
    target_tensor = Variable(torch.zeros(catlen))
    idx_cat = cat_to_ix[target]
    target_tensor[idx_cat]  = 1
    words_tensor = Variable(line_to_tensor(text))
    return target, text, target_tensor, words_tensor

target, text, t_tensor, w_tensor = random_training_example(t
```

- We return the class, the actual text
- And also the matrix representation of these two parts

our RNN

```
import torch.nn as nn
from torch.autograd import Variable

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size)
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size,
                             hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size,
                             output_size)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        return output, hidden
```

Train on one example

```
optimiser = optim.SGD(rnn.parameters(), lr=0.01,
                      momentum=0.9)
criterion = nn.CrossEntropyLoss()
learning_rate = 0.005
def train(target_tensor, words_tensor):
    hidden = rnn.init_hidden()
    rnn.zero_grad()
    for i in range(words_tensor.size()[0]):
        output, hidden = rnn(words_tensor[i], hidden)
    loss = criterion(output.squeeze(),
                     target_tensor.type(torch.LongTensor))
    loss.backward() #magic
    optimiser.step()
    for p in rnn.parameters():
        #need to figure out why this is necessary
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0]
```

Training in a Loop

```
n_iters = 10000

for iter in range(1, n_iters + 1):
    category, line, category_tensor,
    line_tensor, numrow = random_training_example(textdf)
        output, loss = train(category_tensor, line_tensor)
        current_loss += loss
```

Inspecting the Running Model

```
# Print iter number, loss, name and guess
if iter % print_every == 0:
    guess, guess_i = category_from_output(output)
    correct = 'Y' if guess == category else 'N (%s)' % c
    print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter /

# Add current loss avg to list of losses
if iter % plot_every == 0:
    all_losses.append(current_loss / plot_every)
    current_loss = 0
```

Problems

- This loops through the data in a non-deterministic order
- We should probably ensure that we go through the data N*epoch times
- Additionally, we need some test data
- Fortunately, we have all of the text data available
- Unfortunately it's late Monday night now, and I won't sleep if I don't stop working :(

Future Work

- Impelement Deconvolutional Nets/other visualisation tools to understand how the models work
- Solve the actual Kaggle problem by using an RNN over my CNN
- Add the text data, image data and structured data to an ensemble and examine overall performance
- Learn more Python

Conclusions

- PyTorch is a powerful framework for matrix computation on the GPU
- It is deeply integrated with Python
- It's not just a set of bindings to C/C++ code
- It is really easy to install (by the standards of DL frameworks)
- You can inspect each stage of the Net really easily (as it's just Python objects)
- No weirdass errors caused by compilation!

Further Reading

- My repo with code (currently non-functional because I need to upload)
- PyTorch tutorials and examples
- `the Docs` (these are unbelievably large)
- The Book (seriously, even if you never use deep learning there's a lot of really good material there)
- Completely unrelated, but this is an amazing book on Python
- You should definitely read it

Papers (horribly incomplete)

- AlexNet - it's amazing how many new things this paper did
- Deconvolutional Nets
- Gneralised Adversarial Networks
- Rethinking Generalisation and Deep Learning
- Deep Reinforcement Learning