PyTorch Tutorials

Release 0.3.0.post4

Sasank, PyTorch contributors

Jan 27, 2018

BEGINNER TUTORIALS

To get started with learning PyTorch, start with our Beginner Tutorials. The *60-minute blitz* is the most common starting point, and gives you a quick introduction to PyTorch. If you like learning by examples, you will like the tutorial *Learning PyTorch with Examples*

If you would like to do the tutorials interactively via IPython / Jupyter, each tutorial has a download link for a Jupyter Notebook and Python source code.

We also provide a lot of high-quality examples covering image classification, unsupervised learning, reinforcement learning, machine translation and many other applications at https://github.com/pytorch/examples/

You can find reference documentation for PyTorch's API and layers at http://docs.pytorch.org or via inline help. If you would like the tutorials section improved, please open a github issue here with your feedback: https://github.com/pytorch/tutorials

CHAPTER

ONE

BEGINNER TUTORIALS

1.1 Deep Learning with PyTorch: A 60 Minute Blitz

Author: Soumith Chintala

Goal of this tutorial:

- Understand PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

This tutorial assumes that you have a basic familiarity of numpy

Note: Make sure you have the torch and torchvision packages installed.

1.1.1 What is PyTorch?

It's a Python based scientific computing package targeted at two sets of audiences:

- A replacement for numpy to use the power of GPUs
- a deep learning research platform that provides maximum flexibility and speed

Getting Started

Tensors

Tensors are similar to numpy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
from __future__ import print_function
import torch
```

Construct a 5x3 matrix, uninitialized:

```
x = torch.Tensor(5, 3)
print(x)
```

Out:

```
0.0000e+00 -3.6893e+19  0.0000e+00
-3.6893e+19  5.6052e-45  4.5569e-41
 1.2725e-24  1.4013e-45  0.0000e+00
 0.0000e+00  0.0000e+00  2.7802e-40
-2.6772e+25 -1.0845e-19 -2.1732e+25
[torch.FloatTensor of size 5x3]
```

Construct a randomly initialized matrix

```
x = torch.rand(5, 3)
print(x)
```

Out:

```
0.8818  0.0995  0.7781
 0.2704  0.2600  0.7674
 0.7916  0.2011  0.2995
 0.1547  0.8147  0.1139
 0.9433  0.7524  0.0743
[torch.FloatTensor of size 5x3]
```

Get its size

```
print(x.size())
```

Out:

```
torch.Size([5, 3])
```

Note: `torch.Size` is in fact a tuple, so it supports the same operations

Operations

There are multiple syntaxes for operations. Let's see addition as an example

Addition: syntax 1

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

Out:

```
1.2154  0.8710  1.6771
 0.7806  0.8434  1.2302
 0.9313  0.5858  0.8665
 0.5415  1.3722  0.3836
 1.7878  1.0775  0.5611
[torch.FloatTensor of size 5x3]
```

Addition: syntax 2

```
print(torch.add(x, y))
```

Out:

```
1.2154  0.8710  1.6771
 0.7806  0.8434  1.2302
 0.9313  0.5858  0.8665
 0.5415  1.3722  0.3836
 1.7878  1.0775  0.5611
[torch.FloatTensor of size 5x3]
```

Addition: giving an output tensor

```
result = torch.Tensor(5, 3)
torch.add(x, y, out=result)
print(result)
```

Out:

```
1.2154  0.8710  1.6771
 0.7806  0.8434  1.2302
 0.9313  0.5858  0.8665
 0.5415  1.3722  0.3836
 1.7878  1.0775  0.5611
[torch.FloatTensor of size 5x3]
```

Addition: in-place

```
# adds x to y
y.add_(x)
print(y)
```

Out:

```
1.2154  0.8710  1.6771
 0.7806  0.8434  1.2302
 0.9313  0.5858  0.8665
 0.5415  1.3722  0.3836
 1.7878  1.0775  0.5611
[torch.FloatTensor of size 5x3]
```

Note: Any operation that mutates a tensor in-place is post-fixed with an _ For example: `x.copy_(y)`, `x.t_()`, will change `x`.

You can use standard numpy-like indexing with all bells and whistles!

```
print(x[:, 1])
```

Out:

```
0.0995
 0.2600
 0.2011
 0.8147
 0.7524
[torch.FloatTensor of size 5]
```

Resizing: If you want to resize/reshape tensor, you can use `torch.view`:

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
print(x.size(), y.size(), z.size())
```

Out:

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

Read later:

100+ Tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, etc are described here

Numpy Bridge

Converting a torch Tensor to a numpy array and vice versa is a breeze.

The torch Tensor and numpy array will share their underlying memory locations, and changing one will change the other.

Converting torch Tensor to numpy Array

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

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

```
b = a.numpy()
print(b)
```

Out:

```
[ 1.  1.  1.  1.  1.]
```

See how the numpy array changed in value.

```
a.add_(1)
print(a)
print(b)
```

Out:

```
2
 2
 2
 2
 2
```

```
[torch.FloatTensor of size 5]

[ 2.  2.  2.  2.  2.]
```

Converting numpy Array to torch Tensor

See how changing the np array changed the torch Tensor automatically

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[ 2.  2.  2.  2.  2.]

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

CUDA Tensors

Tensors can be moved onto GPU using the `.cuda` function.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

Total running time of the script: (0 minutes 0.004 seconds)

`Download Python source code: tensor_tutorial.py`

`Download Jupyter notebook: tensor_tutorial.ipynb`

Generated by Sphinx-Gallery

1.1.2 Autograd: automatic differentiation

Central to all neural networks in PyTorch is the `autograd` package. Let's first briefly visit this, and we will then go to training our first neural network.

The `autograd` package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

Let us see this in more simple terms with some examples.

Variable

`autograd.Variable` is the central class of the package. It wraps a Tensor, and supports nearly all of operations defined on it. Once you finish your computation you can call `.backward()` and have all the gradients computed automatically.

You can access the raw tensor through the `.data` attribute, while the gradient w.r.t. this variable is accumulated into `.grad`.

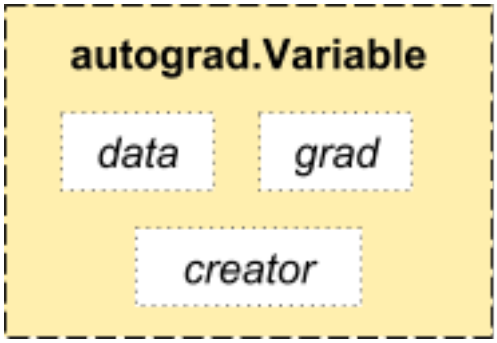


Fig. 1.1: Variable

There's one more class which is very important for autograd implementation - a `Function`.

`Variable` and `Function` are interconnected and build up an acyclic graph, that encodes a complete history of computation. Each variable has a `.grad_fn` attribute that references a `Function` that has created the `Variable` (except for Variables created by the user - their `grad_fn is None`).

If you want to compute the derivatives, you can call `.backward()` on a `Variable`. If `Variable` is a scalar (i.e. it holds a one element data), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `grad_output` argument that is a tensor of matching shape.

```
import torch
from torch.autograd import Variable
```

Create a variable:

```
x = Variable(torch.ones(2, 2), requires_grad=True)
print(x)
```

Out:

```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

Do an operation of variable:

```
y = x + 2
print(y)
```

Out:

```
Variable containing:
 3  3
```

```
 3  3
[torch.FloatTensor of size 2x2]
```

y was created as a result of an operation, so it has a grad_fn.

```
print(y.grad_fn)
```

Out:

```
<AddBackward0 object at 0x10ea01ba8>
```

Do more operations on y

```
z = y * y * 3
out = z.mean()

print(z, out)
```

Out:

```
Variable containing:
 27  27
 27  27
[torch.FloatTensor of size 2x2]
 Variable containing:
 27
[torch.FloatTensor of size 1]
```

Gradients

let's backprop now `out.backward()` is equivalent to doing `out.backward(torch.Tensor([1.0]))`

```
out.backward()
```

print gradients d(out)/dx

```
print(x.grad)
```

Out:

```
Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

You should have got a matrix of `4.5`. Let's call the `out` *Variable* "*o*". We have that $o = \frac{1}{4}\sum_i z_i$, $z_i = 3(x_i + 2)^2$ and $z_i\big|_{x_i=1} = 27$. Therefore, $\frac{\partial o}{\partial x_i} = \frac{3}{2}(x_i + 2)$, hence $\frac{\partial o}{\partial x_i}\big|_{x_i=1} = \frac{9}{2} = 4.5$.

You can do many crazy things with autograd!

```
x = torch.randn(3)
x = Variable(x, requires_grad=True)

y = x * 2
while y.data.norm() < 1000:
    y = y * 2
```

```
print(y)
```

Out:

```
Variable containing:
  206.5757
  146.5951
-1247.5491
[torch.FloatTensor of size 3]
```

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

print(x.grad)
```

Out:

```
Variable containing:
  51.2000
 512.0000
   0.0512
[torch.FloatTensor of size 3]
```

Read Later:

Documentation of `Variable` and `Function` is at http://pytorch.org/docs/autograd

Total running time of the script: (0 minutes 0.007 seconds)

`Download Python source code: autograd_tutorial.py`

`Download Jupyter notebook: autograd_tutorial.ipynb`

Generated by Sphinx-Gallery

1.1.3 Neural Networks

Neural networks can be constructed using the `torch.nn` package.

Now that you had a glimpse of `autograd`, `nn` depends on `autograd` to define models and differentiate them. An `nn.Module` contains layers, and a method `forward(input)` that returns the `output`.

For example, look at this network that classfies digit images:

It is a simple feed-forward network. It takes the input, feeds it through several layers one after the other, and then finally gives the output.

A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: `weight = weight - learning_rate * gradient`

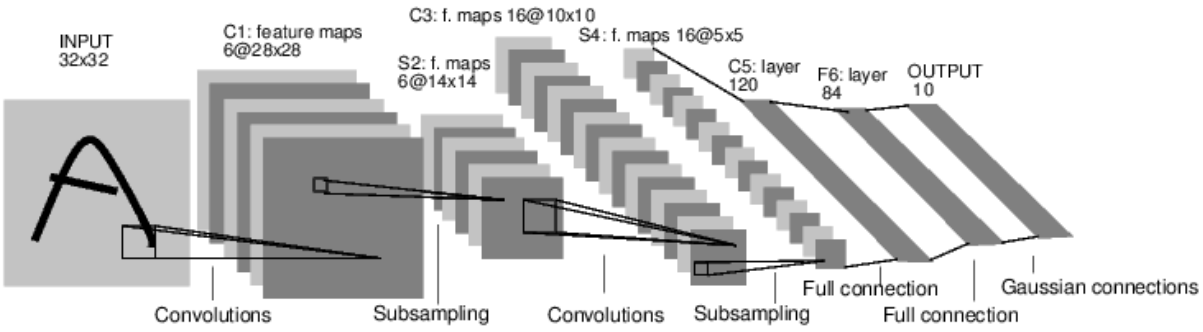


Fig. 1.2: convnet

Define the network

Let's define this network:

```
import torch
from torch.autograd import Variable
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, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 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):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
```

```
print(net)
```

Out:

```
Net(
  (conv1): Conv2d (1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d (6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120)
  (fc2): Linear(in_features=120, out_features=84)
  (fc3): Linear(in_features=84, out_features=10)
)
```

You just have to define the `forward` function, and the `backward` function (where gradients are computed) is automatically defined for you using `autograd`. You can use any of the Tensor operations in the `forward` function.

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
```

Out:

```
10
torch.Size([6, 1, 5, 5])
```

The input to the forward is an `autograd.Variable`, and so is the output. Note: Expected input size to this net(LeNet) is 32x32. To use this net on MNIST dataset,please resize the images from the dataset to 32x32.

```
input = Variable(torch.randn(1, 1, 32, 32))
out = net(input)
print(out)
```

Out:

```
Variable containing:
-0.0456 -0.0181 -0.0920  0.0096  0.1111 -0.0572  0.1209 -0.0945  0.1165 -0.0630
[torch.FloatTensor of size 1x10]
```

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

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

Note: `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.

For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples x nChannels x Height x Width`.

If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

Before proceeding further, let's recap all the classes you've seen so far.

Recap:

- `torch.Tensor` - A *multi-dimensional array*.

- `autograd.Variable` - *Wraps a Tensor and records the history of operations* applied to it. Has the same API as a `Tensor`, with some additions like `backward()`. Also *holds the gradient* w.r.t. the tensor.
- `nn.Module` - Neural network module. *Convenient way of encapsulating parameters*, with helpers for moving them to GPU, exporting, loading, etc.
- `nn.Parameter` - A kind of Variable, that is *automatically registered as a parameter when assigned as an attribute to a* `Module`.
- `autograd.Function` - Implements *forward and backward definitions of an autograd operation*. Every `Variable` operation, creates at least a single `Function` node, that connects to functions that created a `Variable` and *encodes its history*.

At this point, we covered:

- Defining a neural network
- Processing inputs and calling backward.

Still Left:

- Computing the loss
- Updating the weights of the network

Loss Function

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

There are several different loss functions under the nn package . A simple loss is: `nn.MSELoss` which computes the mean-squared error between the input and the target.

For example:

```
output = net(input)
target = Variable(torch.arange(1, 11))  # a dummy target, for example
criterion = nn.MSELoss()

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

Out:

```
Variable containing:
 38.4265
[torch.FloatTensor of size 1]
```

Now, if you follow `loss` in the backward direction, using it's `.grad_fn` attribute, you will see a graph of computations that looks like this:

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

So, when we call `loss.backward()`, the whole graph is differentiated w.r.t. the loss, and all Variables in the graph will have their `.grad` Variable accumulated with the gradient.

For illustration, let us follow a few steps backward:

```
print(loss.grad_fn)  # MSELoss
print(loss.grad_fn.next_functions[0][0])  # Linear
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  # ReLU
```

Out:

```
<MseLossBackward object at 0x10f0ed2e8>
<AddmmBackward object at 0x10f0d4278>
<ExpandBackward object at 0x10f0d4278>
```

Backprop

To backpropagate the error all we have to do is to `loss.backward()`. You need to clear the existing gradients though, else gradients will be accumulated to existing gradients

Now we shall call `loss.backward()`, and have a look at conv1's bias gradients before and after the backward.

```
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)
```

Out:

```
conv1.bias.grad before backward
Variable containing:
 0
 0
 0
 0
 0
 0
[torch.FloatTensor of size 6]

conv1.bias.grad after backward
Variable containing:
1.00000e-02 *
  6.1096
 -9.0981
  2.4990
 -2.4327
 -2.5808
 -2.8642
[torch.FloatTensor of size 6]
```

Now, we have seen how to use loss functions.

Read Later:

> The neural network package contains various modules and loss functions that form the building blocks of deep neural networks. A full list with documentation is here

The only thing left to learn is:

- updating the weights of the network

Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

We can implement this using simple python code:

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)
```

However, as you use neural networks, you want to use various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, we built a small package: `torch.optim` that implements all these methods. Using it is very simple:

```
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
```

Note: Observe how gradient buffers had to be manually set to zero using `optimizer.zero_grad()`. This is because gradients are accumulated as explained in *Backprop* section.

Total running time of the script: (0 minutes 0.013 seconds)

`Download Python source code: neural_networks_tutorial.py`

`Download Jupyter notebook: neural_networks_tutorial.ipynb`

Generated by Sphinx-Gallery

1.1.4 Training a classifier

This is it. You have seen how to define neural networks, compute loss and make updates to the weights of the network.

Now you might be thinking,

What about data?

Generally, when you have to deal with image, text, audio or video data, you can use standard python packages that load data into a numpy array. Then you can convert this array into a `torch.*Tensor`.

- For images, packages such as Pillow, OpenCV are useful.
- For audio, packages such as scipy and librosa

- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful.

Specifically for `vision`, we have created a package called `torchvision`, that has data loaders for common datasets such as Imagenet, CIFAR10, MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.DataLoader`.

This provides a huge convenience and avoids writing boilerplate code.

For this tutorial, we will use the CIFAR10 dataset. It has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.

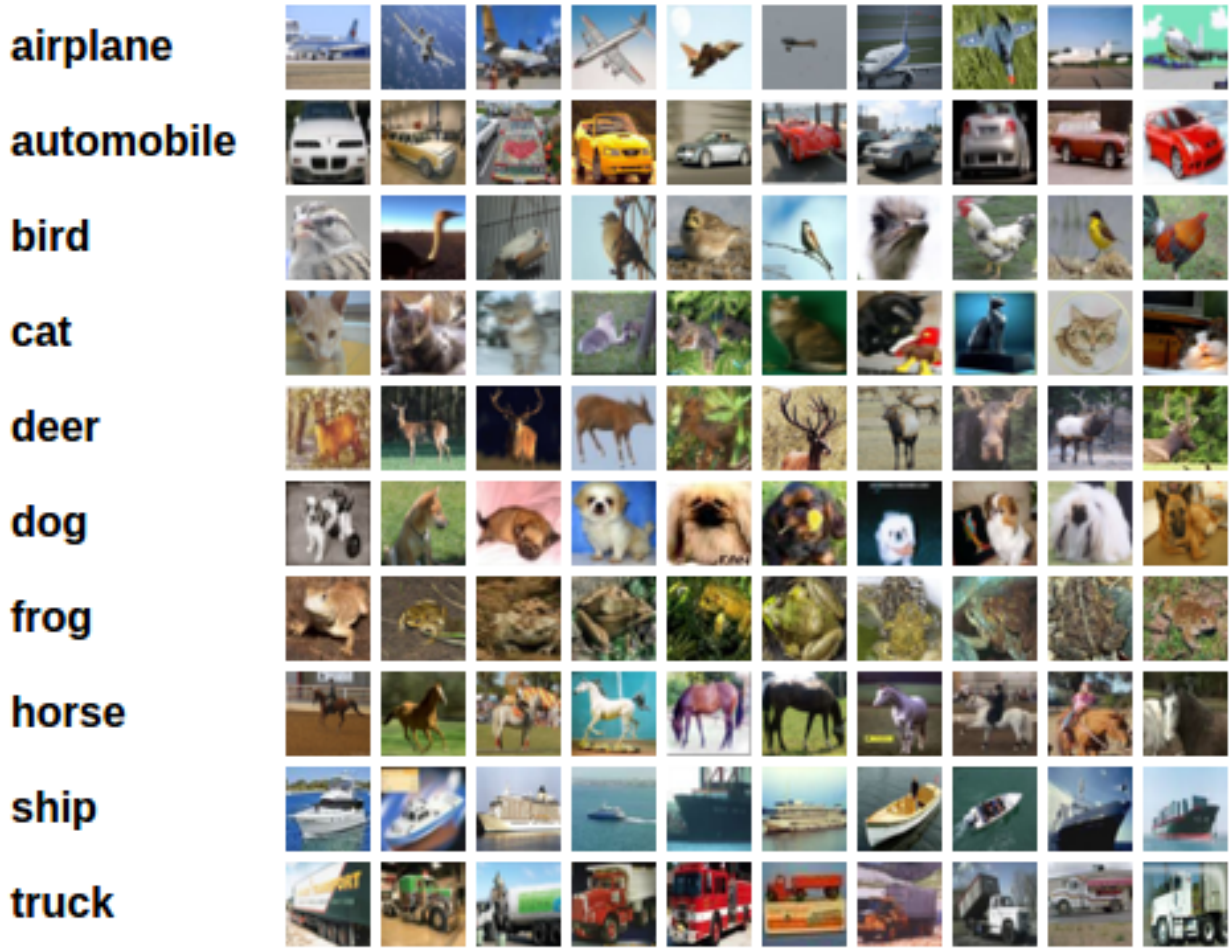


Fig. 1.3: cifar10

Training an image classifier

We will do the following steps in order:

1. Load and normalizing the CIFAR10 training and test datasets using `torchvision`
2. Define a Convolution Neural Network
3. Define a loss function
4. Train the network on the training data

5. Test the network on the test data

1. Loading and normalizing CIFAR10

Using `torchvision`, it's extremely easy to load CIFAR10.

```
import torch
import torchvision
import torchvision.transforms as transforms
```

The output of torchvision datasets are PILImage images of range [0, 1]. We transform them to Tensors of normalized range [-1, 1]

```
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')
```

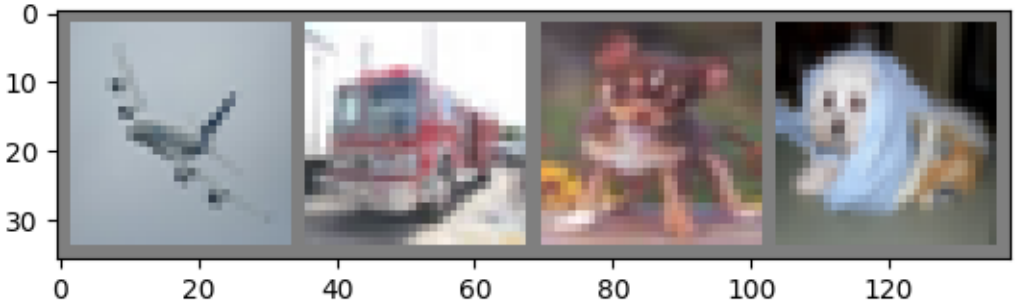
Out:

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-
↪10-python.tar.gz
Files already downloaded and verified
```

Let us show some of the training images, for fun.

```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
```

```
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



Out:

```
plane truck   dog   dog
```

2. Define a Convolution Neural Network

Copy the neural network from the Neural Networks section before and modify it to take 3-channel images (instead of 1-channel images as it was defined).

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


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


net = Net()
```

3. Define a Loss function and optimizer

Let's use a Classification Cross-Entropy loss and SGD with momentum

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

4. Train the network

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the inputs to the network and optimize

```
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
        inputs, labels = data

        # wrap them in Variable
        inputs, labels = Variable(inputs), Variable(labels)

        # 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.data[0]
        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')
```

Out:

```
[1,  2000] loss: 2.210
[1,  4000] loss: 1.891
[1,  6000] loss: 1.691
[1,  8000] loss: 1.604
[1, 10000] loss: 1.500
[1, 12000] loss: 1.478
[2,  2000] loss: 1.412
[2,  4000] loss: 1.377
[2,  6000] loss: 1.341
[2,  8000] loss: 1.337
[2, 10000] loss: 1.301
[2, 12000] loss: 1.277
Finished Training
```

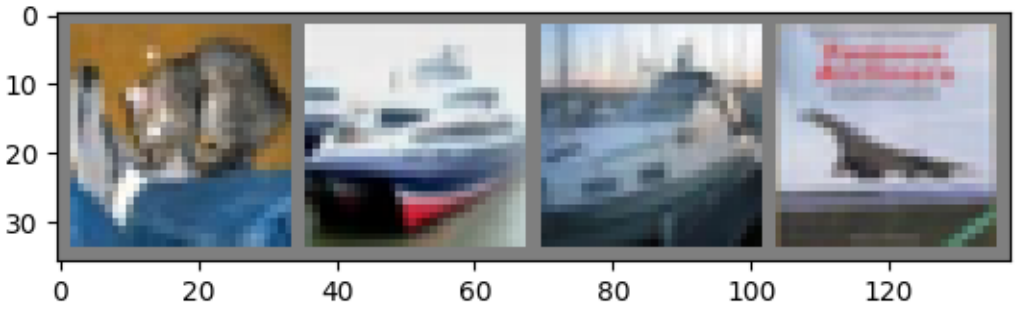
5. Test the network on the test data

We have trained the network for 2 passes over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

```
dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```



Out:

```
GroundTruth:    cat  ship  ship plane
```

Okay, now let us see what the neural network thinks these examples above are:

```
outputs = net(Variable(images))
```

The outputs are energies for the 10 classes. Higher the energy for a class, the more the network thinks that the image is of the particular class. So, let's get the index of the highest energy:

```
_, predicted = torch.max(outputs.data, 1)

print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
                              for j in range(4)))
```

Out:

```
Predicted:    cat  ship   car plane
```

The results seem pretty good.

Let us look at how the network performs on the whole dataset.

```
correct = 0
total = 0
```

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

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

Out:

```
Accuracy of the network on the 10000 test images: 54 %
```

That looks waaay better than chance, which is 10% accuracy (randomly picking a class out of 10 classes). Seems like the network learnt something.

Hmmm, what are the classes that performed well, and the classes that did not perform well:

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
for data in testloader:
    images, labels = data
    outputs = net(Variable(images))
    _, predicted = torch.max(outputs.data, 1)
    c = (predicted == labels).squeeze()
    for i in range(4):
        label = labels[i]
        class_correct[label] += c[i]
        class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

Out:

```
Accuracy of plane : 61 %
Accuracy of   car : 76 %
Accuracy of  bird : 31 %
Accuracy of   cat : 47 %
Accuracy of  deer : 60 %
Accuracy of   dog : 33 %
Accuracy of  frog : 44 %
Accuracy of horse : 66 %
Accuracy of  ship : 66 %
Accuracy of truck : 59 %
```

Okay, so what next?

How do we run these neural networks on the GPU?

Training on GPU

Just like how you transfer a Tensor on to the GPU, you transfer the neural net onto the GPU. This will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

```
net.cuda()
```

Remember that you will have to send the inputs and targets at every step to the GPU too:

```
inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
```

Why dont I notice MASSIVE speedup compared to CPU? Because your network is reallly small.

Exercise: Try increasing the width of your network (argument 2 of the first `nn.Conv2d`, and argument 1 of the second `nn.Conv2d` – they need to be the same number), see what kind of speedup you get.

Goals achieved:

- Understanding PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

Training on multiple GPUs

If you want to see even more MASSIVE speedup using all of your GPUs, please check out *Optional: Data Parallelism*.

Where do I go next?

- *Train neural nets to play video games*
- Train a state-of-the-art ResNet network on imagenet
- Train a face generator using Generative Adversarial Networks
- Train a word-level language model using Recurrent LSTM networks
- More examples
- More tutorials
- Discuss PyTorch on the Forums
- Chat with other users on Slack

Total running time of the script: (7 minutes 40.616 seconds)

`Download Python source code: cifar10_tutorial.py`

`Download Jupyter notebook: cifar10_tutorial.ipynb`

Generated by Sphinx-Gallery

1.1.5 Optional: Data Parallelism

Authors: Sung Kim and Jenny Kang

In this tutorial, we will learn how to use multiple GPUs using `DataParallel`.

It's very easy to use GPUs with PyTorch. You can put the model on a GPU:

```
model.gpu()
```

Then, you can copy all your tensors to the GPU:

```
mytensor = my_tensor.gpu()
```

Please note that just calling `mytensor.gpu()` won't copy the tensor to the GPU. You need to assign it to a new tensor and use that tensor on the GPU.

It's natural to execute your forward, backward propagations on multiple GPUs. However, Pytorch will only use one GPU by default. You can easily run your operations on multiple GPUs by making your model run parallelly using `DataParallel`:

```
model = nn.DataParallel(model)
```

That's the core behind this tutorial. We will explore it in more detail below.

Imports and parameters

Import PyTorch modules and define parameters.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.utils.data import Dataset, DataLoader

# Parameters and DataLoaders
input_size = 5
output_size = 2

batch_size = 30
data_size = 100
```

Dummy DataSet

Make a dummy (random) dataset. You just need to implement the getitem

```
class RandomDataset(Dataset):

    def __init__(self, size, length):
        self.len = length
        self.data = torch.randn(length, size)

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return self.len

rand_loader = DataLoader(dataset=RandomDataset(input_size, 100),
                         batch_size=batch_size, shuffle=True)
```

Simple Model

For the demo, our model just gets an input, performs a linear operation, and gives an output. However, you can use `DataParallel` on any model (CNN, RNN, Capsule Net etc.)

We've placed a print statement inside the model to monitor the size of input and output tensors. Please pay attention to what is printed at batch rank 0.

```
class Model(nn.Module):
    # Our model

    def __init__(self, input_size, output_size):
        super(Model, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, input):
        output = self.fc(input)
        print("  In Model: input size", input.size(),
              "output size", output.size())

        return output
```

Create Model and DataParallel

This is the core part of the tutorial. First, we need to make a model instance and check if we have multiple GPUs. If we have multiple GPUs, we can wrap our model using `nn.DataParallel`. Then we can put our model on GPUs by `model.gpu()`

```
model = Model(input_size, output_size)
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
  model = nn.DataParallel(model)

if torch.cuda.is_available():
   model.cuda()
```

Run the Model

Now we can see the sizes of input and output tensors.

```
for data in rand_loader:
    if torch.cuda.is_available():
        input_var = Variable(data.cuda())
    else:
        input_var = Variable(data)

    output = model(input_var)
    print("Outside: input size", input_var.size(),
          "output_size", output.size())
```

Out:

```
In Model: input size torch.Size([30, 5]) output size torch.Size([30, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
  In Model: input size torch.Size([30, 5]) output size torch.Size([30, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
  In Model: input size torch.Size([30, 5]) output size torch.Size([30, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
  In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

Results

When we batch 30 inputs and 30 outputs, the model gets 30 and outputs 30 as expected. But if you have GPUs, then you can get results like this.

2 GPUs

If you have 2, you will see:

```
# on 2 GPUs
Let's use 2 GPUs!
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
    In Model: input size torch.Size([15, 5]) output size torch.Size([15, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
    In Model: input size torch.Size([5, 5]) output size torch.Size([5, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

3 GPUs

If you have 3 GPUs, you will see:

```
Let's use 3 GPUs!
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
    In Model: input size torch.Size([10, 5]) output size torch.Size([10, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

8 GPUs

If you have 8, you will see:

```
Let's use 8 GPUs!
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([4, 5]) output size torch.Size([4, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([30, 5]) output_size torch.Size([30, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
    In Model: input size torch.Size([2, 5]) output size torch.Size([2, 2])
Outside: input size torch.Size([10, 5]) output_size torch.Size([10, 2])
```

Summary

DataParallel splits your data automatically and sends job orders to multiple models on several GPUs. After each model finishes their job, DataParallel collects and merges the results before returning it to you.

For more information, please check out http://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html.

Total running time of the script: (0 minutes 0.002 seconds)

`Download Python source code: data_parallel_tutorial.py`

`Download Jupyter notebook: data_parallel_tutorial.ipynb`

Generated by Sphinx-Gallery

1.2 PyTorch for former Torch users

Author: Soumith Chintala

In this tutorial, you will learn the following:

1. Using torch Tensors, and important difference against (Lua)Torch
2. Using the autograd package
3. Building neural networks
   - Building a ConvNet
   - Building a Recurrent Net
4. Use multiple GPUs

1.2.1 Tensors

Tensors behave almost exactly the same way in PyTorch as they do in Torch.

Create a tensor of size (5 x 7) with uninitialized memory:

```
import torch
a = torch.FloatTensor(5, 7)
```

Initialize a tensor randomized with a normal distribution with mean=0, var=1:

```
a = torch.randn(5, 7)
print(a)
print(a.size())
```

Out:

```
-0.0927 -0.7228 -0.0720 -1.0393  0.5796  0.9966  1.4888
-0.8248  1.3182 -0.3825 -0.4135 -0.0876 -0.4456 -0.4691
 1.3551  2.3201  0.9926 -1.2327 -0.6083 -0.7663 -0.8656
 1.4475  0.1751 -0.3004  0.4564 -2.5295 -0.3284 -0.6914
 0.2594 -1.9741 -0.1869 -0.6838 -2.1483  1.4089 -0.1694
[torch.FloatTensor of size 5x7]

torch.Size([5, 7])
```

Note: `torch.Size` is in fact a tuple, so it supports the same operations

Inplace / Out-of-place

The first difference is that ALL operations on the tensor that operate in-place on it will have an _ postfix. For example, `add` is the out-of-place version, and `add_` is the in-place version.

```
a.fill_(3.5)
# a has now been filled with the value 3.5

b = a.add(4.0)
# a is still filled with 3.5
# new tensor b is returned with values 3.5 + 4.0 = 7.5

print(a, b)
```

Out:

```
3.5000  3.5000  3.5000  3.5000  3.5000  3.5000  3.5000
 3.5000  3.5000  3.5000  3.5000  3.5000  3.5000  3.5000
 3.5000  3.5000  3.5000  3.5000  3.5000  3.5000  3.5000
 3.5000  3.5000  3.5000  3.5000  3.5000  3.5000  3.5000
 3.5000  3.5000  3.5000  3.5000  3.5000  3.5000  3.5000
[torch.FloatTensor of size 5x7]

 7.5000  7.5000  7.5000  7.5000  7.5000  7.5000  7.5000
 7.5000  7.5000  7.5000  7.5000  7.5000  7.5000  7.5000
 7.5000  7.5000  7.5000  7.5000  7.5000  7.5000  7.5000
 7.5000  7.5000  7.5000  7.5000  7.5000  7.5000  7.5000
 7.5000  7.5000  7.5000  7.5000  7.5000  7.5000  7.5000
[torch.FloatTensor of size 5x7]
```

Some operations like `narrow` do not have in-place versions, and hence, `.narrow_` does not exist. Similarly, some operations like `fill_` do not have an out-of-place version, so `.fill` does not exist.

Zero Indexing

Another difference is that Tensors are zero-indexed. (In lua, tensors are one-indexed)

```
b = a[0, 3]  # select 1st row, 4th column from a
```

Tensors can be also indexed with Python's slicing

```
b = a[:, 3:5]  # selects all rows, 4th column and  5th column from a
```

No camel casing

The next small difference is that all functions are now NOT camelCase anymore. For example `indexAdd` is now called `index_add_`

```
x = torch.ones(5, 5)
print(x)
```

Out:

```
1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
 1  1  1  1  1
[torch.FloatTensor of size 5x5]
```

```
z = torch.Tensor(5, 2)
z[:, 0] = 10
z[:, 1] = 100
print(z)
```

Out:

```
10  100
  10  100
  10  100
  10  100
```

```
  10  100
[torch.FloatTensor of size 5x2]
```

```
x.index_add_(1, torch.LongTensor([4, 0]), z)
print(x)
```

Out:

```
101    1    1    1   11
 101    1    1    1   11
 101    1    1    1   11
 101    1    1    1   11
 101    1    1    1   11
[torch.FloatTensor of size 5x5]
```

Numpy Bridge

Converting a torch Tensor to a numpy array and vice versa is a breeze. The torch Tensor and numpy array will share their underlying memory locations, and changing one will change the other.

Converting torch Tensor to numpy Array

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

Out:

```
1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

```
b = a.numpy()
print(b)
```

Out:

```
[ 1.  1.  1.  1.  1.]
```

```
a.add_(1)
print(a)
print(b)    # see how the numpy array changed in value
```

Out:

```
2
 2
 2
 2
 2
[torch.FloatTensor of size 5]
```

```
[ 2.  2.  2.  2.  2.]
```

Converting numpy Array to torch Tensor

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)  # see how changing the np array changed the torch Tensor automatically
```

Out:

```
[ 2.  2.  2.  2.  2.]

 2
 2
 2
 2
 2
[torch.DoubleTensor of size 5]
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.

CUDA Tensors

CUDA Tensors are nice and easy in pytorch, and transfering a CUDA tensor from the CPU to GPU will retain its underlying type.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    # creates a LongTensor and transfers it
    # to GPU as torch.cuda.LongTensor
    a = torch.LongTensor(10).fill_(3).cuda()
    print(type(a))
    b = a.cpu()
    # transfers it to CPU, back to
    # being a torch.LongTensor
```

Total running time of the script: (0 minutes 0.003 seconds)

Download Python source code: tensor_tutorial.py

Download Jupyter notebook: tensor_tutorial.ipynb

Generated by Sphinx-Gallery

1.2.2 Autograd

Autograd is now a core torch package for automatic differentiation. It uses a tape based system for automatic differentiation.

In the forward phase, the autograd tape will remember all the operations it executed, and in the backward phase, it will replay the operations.

Variable

In autograd, we introduce a `Variable` class, which is a very thin wrapper around a `Tensor`. You can access the raw tensor through the `.data` attribute, and after computing the backward pass, a gradient w.r.t. this variable is accumulated into `.grad` attribute.

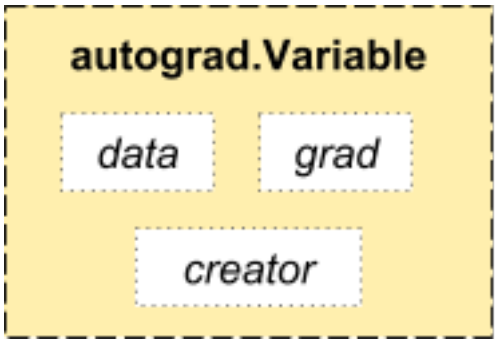


Fig. 1.4: Variable

There's one more class which is very important for autograd implementation - a `Function`. `Variable` and `Function` are interconnected and build up an acyclic graph, that encodes a complete history of computation. Each variable has a `.grad_fn` attribute that references a function that has created a function (except for Variables created by the user - these have `None` as `.grad_fn`).

If you want to compute the derivatives, you can call `.backward()` on a `Variable`. If `Variable` is a scalar (i.e. it holds a one element tensor), you don't need to specify any arguments to `backward()`, however if it has more elements, you need to specify a `grad_output` argument that is a tensor of matching shape.

```
import torch
from torch.autograd import Variable
x = Variable(torch.ones(2, 2), requires_grad=True)
print(x)  # notice the "Variable containing" line
```

Out:

```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

```
print(x.data)
```

Out:

```
1  1
 1  1
[torch.FloatTensor of size 2x2]
```

```
print(x.grad)
```

Out:

```
None
```

```
print(x.grad_fn)  # we've created x ourselves
```

Out:

```
None
```

Do an operation of x:

```
y = x + 2
print(y)
```

Out:

```
Variable containing:
 3  3
 3  3
[torch.FloatTensor of size 2x2]
```

y was created as a result of an operation, so it has a grad_fn

```
print(y.grad_fn)
```

Out:

```
<AddBackward0 object at 0x125e9aac8>
```

More operations on y:

```
z = y * y * 3
out = z.mean()

print(z, out)
```

Out:

```
Variable containing:
 27  27
 27  27
[torch.FloatTensor of size 2x2]
 Variable containing:
 27
[torch.FloatTensor of size 1]
```

Gradients

let's backprop now and print gradients d(out)/dx

```
out.backward()
print(x.grad)
```

Out:

```
Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

By default, gradient computation flushes all the internal buffers contained in the graph, so if you even want to do the backward on some part of the graph twice, you need to pass in `retain_variables = True` during the first pass.

```
x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 2
y.backward(torch.ones(2, 2), retain_graph=True)
# the retain_variables flag will prevent the internal buffers from being freed
print(x.grad)
```

Out:

```
Variable containing:
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

```
z = y * y
print(z)
```

Out:

```
Variable containing:
 9  9
 9  9
[torch.FloatTensor of size 2x2]
```

just backprop random gradients

```
gradient = torch.randn(2, 2)

# this would fail if we didn't specify
# that we want to retain variables
y.backward(gradient)

print(x.grad)
```

Out:

```
Variable containing:
 1.6855  1.1999
 0.9893  1.2909
[torch.FloatTensor of size 2x2]
```

Total running time of the script: (0 minutes 0.002 seconds)

`Download Python source code: autograd_tutorial.py`

`Download Jupyter notebook: autograd_tutorial.ipynb`

Generated by Sphinx-Gallery

1.2.3 nn package

We've redesigned the nn package, so that it's fully integrated with autograd. Let's review the changes.

Replace containers with autograd:

> You no longer have to use Containers like `ConcatTable`, or modules like `CAddTable`, or use and debug with nngraph. We will seamlessly use autograd to define our neural networks. For example,

- `output = nn.CAddTable():forward({input1, input2})` simply becomes `output = input1 + input2`
- `output = nn.MulConstant(0.5):forward(input)` simply becomes `output = input * 0.5`

State is no longer held in the module, but in the network graph:

Using recurrent networks should be simpler because of this reason. If you want to create a recurrent network, simply use the same Linear layer multiple times, without having to think about sharing weights.

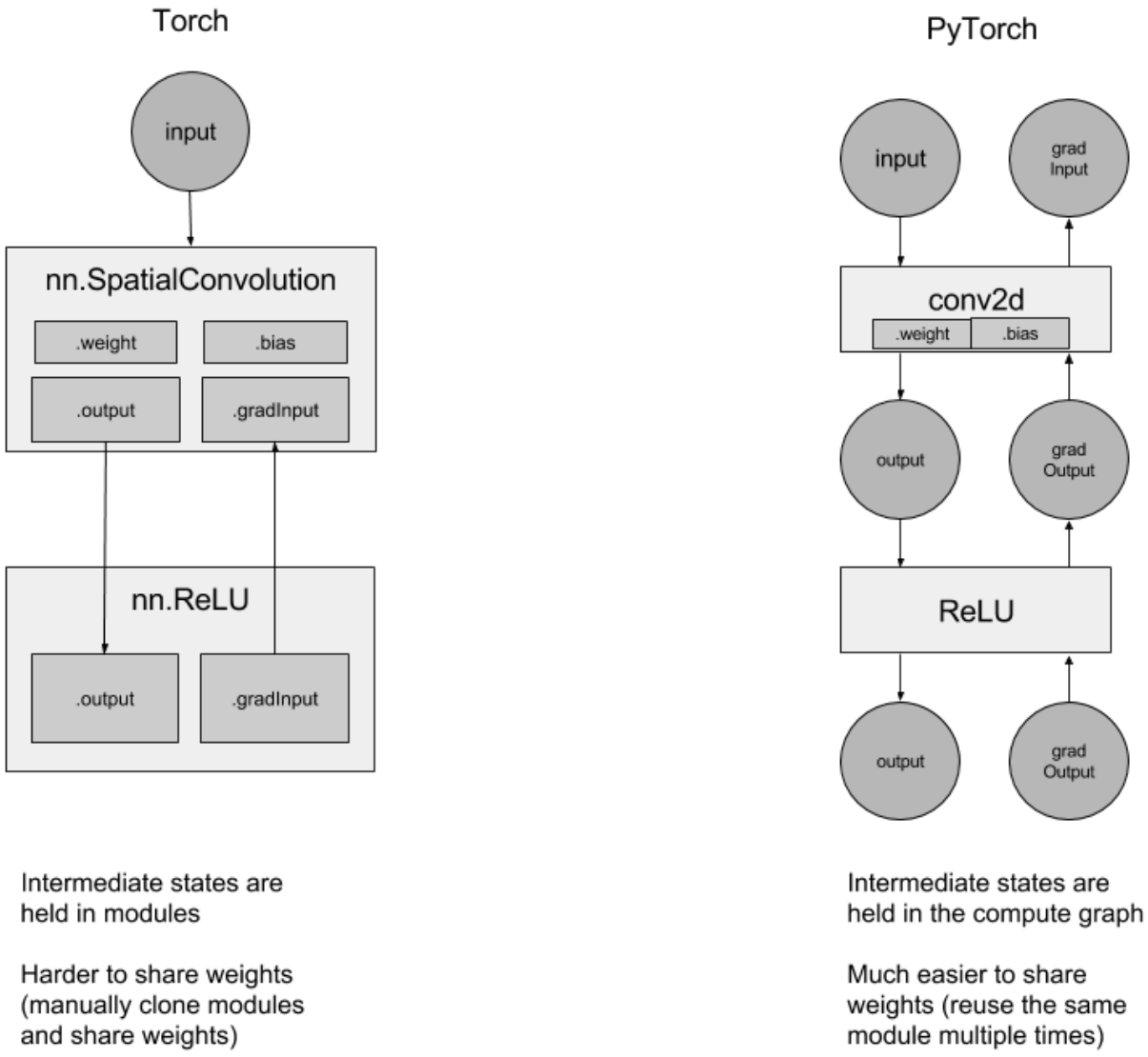


Fig. 1.5: torch-nn-vs-pytorch-nn

Simplified debugging:

Debugging is intuitive using Python's pdb debugger, and **the debugger and stack traces stop at exactly where an error occurred.** What you see is what you get.

Example 1: ConvNet

Let's see how to create a small ConvNet.

All of your networks are derived from the base class `nn.Module`:

- In the constructor, you declare all the layers you want to use.
- In the forward function, you define how your model is going to be run, from input to output

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


class MNISTConvNet(nn.Module):

    def __init__(self):
        # this is the place where you instantiate all your modules
        # you can later access them using the same names you've given them in
        # here
        super(MNISTConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(10, 20, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    # it's the forward function that defines the network structure
    # we're accepting only a single input in here, but if you want,
    # feel free to use more
    def forward(self, input):
        x = self.pool1(F.relu(self.conv1(input)))
        x = self.pool2(F.relu(self.conv2(x)))

        # in your model definition you can go full crazy and use arbitrary
        # python code to define your model structure
        # all these are perfectly legal, and will be handled correctly
        # by autograd:
        # if x.gt(0) > x.numel() / 2:
        #      ...
        #
        # you can even do a loop and reuse the same module inside it
        # modules no longer hold ephemeral state, so you can use them
        # multiple times during your forward pass
        # while x.norm(2) < 10:
        #    x = self.conv1(x)

        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x
```

Let's use the defined ConvNet now. You create an instance of the class first.

```
net = MNISTConvNet()
print(net)
```

Out:

```
MNISTConvNet(
  (conv1): Conv2d (1, 10, kernel_size=(5, 5), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (conv2): Conv2d (10, 20, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1))
  (fc1): Linear(in_features=320, out_features=50)
  (fc2): Linear(in_features=50, out_features=10)
)
```

Note: `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.

For example, `nn.Conv2d` will take in a 4D Tensor of `nSamples x nChannels x Height x Width`.

If you have a single sample, just use `input.unsqueeze(0)` to add a fake batch dimension.

Create a mini-batch containing a single sample of random data and send the sample through the ConvNet.

```
input = Variable(torch.randn(1, 1, 28, 28))
out = net(input)
print(out.size())
```

Out:

```
torch.Size([1, 10])
```

Define a dummy target label and compute error using a loss function.

```
target = Variable(torch.LongTensor([3]))
loss_fn = nn.CrossEntropyLoss()  # LogSoftmax + ClassNLL Loss
err = loss_fn(out, target)
err.backward()

print(err)
```

Out:

```
Variable containing:
 2.2420
[torch.FloatTensor of size 1]
```

The output of the ConvNet `out` is a `Variable`. We compute the loss using that, and that results in `err` which is also a `Variable`. Calling `.backward` on `err` hence will propagate gradients all the way through the ConvNet to it's weights

Let's access individual layer weights and gradients:

```
print(net.conv1.weight.grad.size())
```

Out:

```
torch.Size([10, 1, 5, 5])
```

```
print(net.conv1.weight.data.norm())  # norm of the weight
print(net.conv1.weight.grad.data.norm())  # norm of the gradients
```

Out:

```
1.7683947144545937
0.5659629674106466
```

Forward and Backward Function Hooks

We've inspected the weights and the gradients. But how about inspecting / modifying the output and grad_output of a layer?

We introduce **hooks** for this purpose.

You can register a function on a `Module` or a `Variable`. The hook can be a forward hook or a backward hook. The forward hook will be executed when a forward call is executed. The backward hook will be executed in the backward phase. Let's look at an example.

We register a forward hook on conv2 and print some information

```
def printnorm(self, input, output):
    # input is a tuple of packed inputs
    # output is a Variable. output.data is the Tensor we are interested
    print('Inside ' + self.__class__.__name__ + ' forward')
    print('')
    print('input: ', type(input))
    print('input[0]: ', type(input[0]))
    print('output: ', type(output))
    print('')
    print('input size:', input[0].size())
    print('output size:', output.data.size())
    print('output norm:', output.data.norm())


net.conv2.register_forward_hook(printnorm)

out = net(input)
```

Out:

```
Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.autograd.variable.Variable'>
output:  <class 'torch.autograd.variable.Variable'>

input size: torch.Size([1, 10, 12, 12])
output size: torch.Size([1, 20, 8, 8])
output norm: 15.615115080240367
```

We register a backward hook on conv2 and print some information

```
def printgradnorm(self, grad_input, grad_output):
    print('Inside ' + self.__class__.__name__ + ' backward')
    print('Inside class:' + self.__class__.__name__)
    print('')
    print('grad_input: ', type(grad_input))
    print('grad_input[0]: ', type(grad_input[0]))
    print('grad_output: ', type(grad_output))
    print('grad_output[0]: ', type(grad_output[0]))
```

```
    print('')
    print('grad_input size:', grad_input[0].size())
    print('grad_output size:', grad_output[0].size())
    print('grad_input norm:', grad_input[0].data.norm())


net.conv2.register_backward_hook(printgradnorm)

out = net(input)
err = loss_fn(out, target)
err.backward()
```

Out:

```
Inside Conv2d forward

input:  <class 'tuple'>
input[0]:  <class 'torch.autograd.variable.Variable'>
output:  <class 'torch.autograd.variable.Variable'>

input size: torch.Size([1, 10, 12, 12])
output size: torch.Size([1, 20, 8, 8])
output norm: 15.615115080240367
Inside Conv2d backward
Inside class:Conv2d

grad_input:  <class 'tuple'>
grad_input[0]:  <class 'torch.autograd.variable.Variable'>
grad_output:  <class 'tuple'>
grad_output[0]:  <class 'torch.autograd.variable.Variable'>

grad_input size: torch.Size([1, 10, 12, 12])
grad_output size: torch.Size([1, 20, 8, 8])
grad_input norm: 0.11870614155106103
```

A full and working MNIST example is located here https://github.com/pytorch/examples/tree/master/mnist

Example 2: Recurrent Net

Next, let's look at building recurrent nets with PyTorch.

Since the state of the network is held in the graph and not in the layers, you can simply create an nn.Linear and reuse it over and over again for the recurrence.

```
class RNN(nn.Module):

    # you can also accept arguments in your model constructor
    def __init__(self, data_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        input_size = data_size + hidden_size

        self.i2h = nn.Linear(input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)

    def forward(self, data, last_hidden):
```

```
        input = torch.cat((data, last_hidden), 1)
        hidden = self.i2h(input)
        output = self.h2o(hidden)
        return hidden, output


rnn = RNN(50, 20, 10)
```

A more complete Language Modeling example using LSTMs and Penn Tree-bank is located here

PyTorch by default has seamless CuDNN integration for ConvNets and Recurrent Nets

```
loss_fn = nn.MSELoss()

batch_size = 10
TIMESTEPS = 5

# Create some fake data
batch = Variable(torch.randn(batch_size, 50))
hidden = Variable(torch.zeros(batch_size, 20))
target = Variable(torch.zeros(batch_size, 10))

loss = 0
for t in range(TIMESTEPS):
    # yes! you can reuse the same network several times,
    # sum up the losses, and call backward!
    hidden, output = rnn(batch, hidden)
    loss += loss_fn(output, target)
loss.backward()
```

Total running time of the script: (0 minutes 0.009 seconds)

`Download Python source code: nn_tutorial.py`

`Download Jupyter notebook: nn_tutorial.ipynb`

Generated by Sphinx-Gallery

1.2.4 Multi-GPU examples

Data Parallelism is when we split the mini-batch of samples into multiple smaller mini-batches and run the computation for each of the smaller mini-batches in parallel.

Data Parallelism is implemented using `torch.nn.DataParallel`. One can wrap a Module in `DataParallel` and it will be parallelized over multiple GPUs in the batch dimension.

DataParallel

```
import torch.nn as nn


class DataParallelModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.block1 = nn.Linear(10, 20)
```

```
        # wrap block2 in DataParallel
        self.block2 = nn.Linear(20, 20)
        self.block2 = nn.DataParallel(self.block2)

        self.block3 = nn.Linear(20, 20)

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        return x
```

The code does not need to be changed in CPU-mode.

The documentation for DataParallel is here.

Primitives on which DataParallel is implemented upon:

In general, pytorch's *nn.parallel* primitives can be used independently. We have implemented simple MPI-like primitives:

- replicate: replicate a Module on multiple devices
- scatter: distribute the input in the first-dimension
- gather: gather and concatenate the input in the first-dimension
- parallel_apply: apply a set of already-distributed inputs to a set of already-distributed models.

To give a better clarity, here function `data_parallel` composed using these collectives

```
def data_parallel(module, input, device_ids, output_device=None):
    if not device_ids:
        return module(input)

    if output_device is None:
        output_device = device_ids[0]

    replicas = nn.parallel.replicate(module, device_ids)
    inputs = nn.parallel.scatter(input, device_ids)
    replicas = replicas[:len(inputs)]
    outputs = nn.parallel.parallel_apply(replicas, inputs)
    return nn.parallel.gather(outputs, output_device)
```

Part of the model on CPU and part on the GPU

Let's look at a small example of implementing a network where part of it is on the CPU and part on the GPU

```
class DistributedModel(nn.Module):

    def __init__(self):
        super().__init__(
            embedding=nn.Embedding(1000, 10),
            rnn=nn.Linear(10, 10).cuda(0),
        )

    def forward(self, x):
        # Compute embedding on CPU
        x = self.embedding(x)
```

```
        # Transfer to GPU
        x = x.cuda(0)

        # Compute RNN on GPU
        x = self.rnn(x)
        return x
```

This was a small introduction to PyTorch for former Torch users. There's a lot more to learn.

Look at our more comprehensive introductory tutorial which introduces the `optim` package, data loaders etc.: *Deep Learning with PyTorch: A 60 Minute Blitz*.

Also look at

- *Train neural nets to play video games*
- Train a state-of-the-art ResNet network on imagenet
- Train an face generator using Generative Adversarial Networks
- Train a word-level language model using Recurrent LSTM networks
- More examples
- More tutorials
- Discuss PyTorch on the Forums
- Chat with other users on Slack

Total running time of the script: (0 minutes 0.001 seconds)

`Download Python source code: parallelism_tutorial.py`

`Download Jupyter notebook: parallelism_tutorial.ipynb`

Generated by Sphinx-Gallery

1.3 Learning PyTorch with Examples

Author: Justin Johnson

This tutorial introduces the fundamental concepts of PyTorch through self-contained examples.

At its core, PyTorch provides two main features:

- An n-dimensional Tensor, similar to numpy but can run on GPUs
- Automatic differentiation for building and training neural networks

We will use a fully-connected ReLU network as our running example. The network will have a single hidden layer, and will be trained with gradient descent to fit random data by minimizing the Euclidean distance between the network output and the true output.

Note: You can browse the individual examples at the *end of this page*.

Table of Contents

1.3.1 Tensors

Warm-up: numpy

Before introducing PyTorch, we will first implement the network using numpy.

Numpy provides an n-dimensional array object, and many functions for manipulating these arrays. Numpy is a generic framework for scientific computing; it does not know anything about computation graphs, or deep learning, or gradients. However we can easily use numpy to fit a two-layer network to random data by manually implementing the forward and backward passes through the network using numpy operations:

```
# -*- coding: utf-8 -*-
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
```

```
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

PyTorch: Tensors

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of 50x or greater, so unfortunately numpy won't be enough for modern deep learning.

Here we introduce the most fundamental PyTorch concept: the **Tensor**. A PyTorch Tensor is conceptually identical to a numpy array: a Tensor is an n-dimensional array, and PyTorch provides many functions for operating on these Tensors. Like numpy arrays, PyTorch Tensors do not know anything about deep learning or computational graphs or gradients; they are a generic tool for scientific computing.

However unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations. To run a PyTorch Tensor on GPU, you simply need to cast it to a new datatype.

Here we use PyTorch Tensors to fit a two-layer network to random data. Like the numpy example above we need to manually implement the forward and backward passes through the network:

```
# -*- coding: utf-8 -*-

import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
```

```
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

1.3.2 Autograd

PyTorch: Variables and autograd

In the above examples, we had to manually implement both the forward and backward passes of our neural network. Manually implementing the backward pass is not a big deal for a small two-layer network, but can quickly get very hairy for large complex networks.

Thankfully, we can use automatic differentiation to automate the computation of backward passes in neural networks. The **autograd** package in PyTorch provides exactly this functionality. When using autograd, the forward pass of your network will define a **computational graph**; nodes in the graph will be Tensors, and edges will be functions that produce output Tensors from input Tensors. Backpropagating through this graph then allows you to easily compute gradients.

This sounds complicated, it's pretty simple to use in practice. We wrap our PyTorch Tensors in **Variable** objects; a Variable represents a node in a computational graph. If `x` is a Variable then `x.data` is a Tensor, and `x.grad` is another Variable holding the gradient of `x` with respect to some scalar value.

PyTorch Variables have the same API as PyTorch Tensors: (almost) any operation that you can perform on a Tensor also works on Variables; the difference is that using Variables defines a computational graph, allowing you to automatically compute gradients.

Here we use PyTorch Variables and autograd to implement our two-layer network; now we no longer need to manually implement the backward pass through the network:

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable

dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
```

```
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs, and wrap them in Variables.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Variables during the backward pass.
x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

# Create random Tensors for weights, and wrap them in Variables.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Variables during the backward pass.
w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Variables; these
    # are exactly the same operations we used to compute the forward pass using
    # Tensors, but we do not need to keep references to intermediate values since
    # we are not implementing the backward pass by hand.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Compute and print loss using operations on Variables.
    # Now loss is a Variable of shape (1,) and loss.data is a Tensor of shape
    # (1,); loss.data[0] is a scalar value holding the loss.
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.data[0])

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Variables with requires_grad=True.
    # After this call w1.grad and w2.grad will be Variables holding the gradient
    # of the loss with respect to w1 and w2 respectively.
    loss.backward()

    # Update weights using gradient descent; w1.data and w2.data are Tensors,
    # w1.grad and w2.grad are Variables and w1.grad.data and w2.grad.data are
    # Tensors.
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data

    # Manually zero the gradients after updating weights
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

PyTorch: Defining new autograd functions

Under the hood, each primitive autograd operator is really two functions that operate on Tensors. The **forward** function computes output Tensors from input Tensors. The **backward** function receives the gradient of the output Tensors with respect to some scalar value, and computes the gradient of the input Tensors with respect to that same scalar value.

In PyTorch we can easily define our own autograd operator by defining a subclass of `torch.autograd.Function` and implementing the `forward` and `backward` functions. We can then use our new autograd operator by constructing an instance and calling it like a function, passing Variables containing input data.

In this example we define our own custom autograd function for performing the ReLU nonlinearity, and use it to implement our two-layer network:

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

# Create random Tensors for weights, and wrap them in Variables.
w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply

    # Forward pass: compute predicted y using operations on Variables; we compute
    # ReLU using our custom autograd operation.
```

```
    y_pred = relu(x.mm(w1)).mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.data[0])

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data

    # Manually zero the gradients after updating weights
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

TensorFlow: Static Graphs

PyTorch autograd looks a lot like TensorFlow: in both frameworks we define a computational graph, and use automatic differentiation to compute gradients. The biggest difference between the two is that TensorFlow's computational graphs are **static** and PyTorch uses **dynamic** computational graphs.

In TensorFlow, we define the computational graph once and then execute the same graph over and over again, possibly feeding different input data to the graph. In PyTorch, each forward pass defines a new computational graph.

Static graphs are nice because you can optimize the graph up front; for example a framework might decide to fuse some graph operations for efficiency, or to come up with a strategy for distributing the graph across many GPUs or many machines. If you are reusing the same graph over and over, then this potentially costly up-front optimization can be amortized as the same graph is rerun over and over.

One aspect where static and dynamic graphs differ is control flow. For some models we may wish to perform different computation for each data point; for example a recurrent network might be unrolled for different numbers of time steps for each data point; this unrolling can be implemented as a loop. With a static graph the loop construct needs to be a part of the graph; for this reason TensorFlow provides operators such as `tf.scan` for embedding loops into the graph. With dynamic graphs the situation is simpler: since we build graphs on-the-fly for each example, we can use normal imperative flow control to perform computation that differs for each input.

To contrast with the PyTorch autograd example above, here we use TensorFlow to fit a simple two-layer net:

```
# -*- coding: utf-8 -*-
import tensorflow as tf
import numpy as np

# First we set up the computational graph:

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create placeholders for the input and target data; these will be filled
# with real data when we execute the graph.
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# Create Variables for the weights and initialize them with random data.
# A TensorFlow Variable persists its value across executions of the graph.
```

```
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# Forward pass: Compute the predicted y using operations on TensorFlow Tensors.
# Note that this code does not actually perform any numeric operations; it
# merely sets up the computational graph that we will later execute.
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

# Compute loss using operations on TensorFlow Tensors
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# Compute gradient of the loss with respect to w1 and w2.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

# Update the weights using gradient descent. To actually update the weights
# we need to evaluate new_w1 and new_w2 when executing the graph. Note that
# in TensorFlow the the act of updating the value of the weights is part of
# the computational graph; in PyTorch this happens outside the computational
# graph.
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

# Now we have built our computational graph, so we enter a TensorFlow session to
# actually execute the graph.
with tf.Session() as sess:
    # Run the graph once to initialize the Variables w1 and w2.
    sess.run(tf.global_variables_initializer())

    # Create numpy arrays holding the actual data for the inputs x and targets
    # y
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        # Execute the graph many times. Each time it executes we want to bind
        # x_value to x and y_value to y, specified with the feed_dict argument.
        # Each time we execute the graph we want to compute the values for loss,
        # new_w1, and new_w2; the values of these Tensors are returned as numpy
        # arrays.
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)
```

1.3.3 *nn* module

PyTorch: nn

Computational graphs and autograd are a very powerful paradigm for defining complex operators and automatically taking derivatives; however for large neural networks raw autograd can be a bit too low-level.

When building neural networks we frequently think of arranging the computation into **layers**, some of which have **learnable parameters** which will be optimized during learning.

In TensorFlow, packages like Keras, TensorFlow-Slim, and TFLearn provide higher-level abstractions over raw com-

putational graphs that are useful for building neural networks.

In PyTorch, the `nn` package serves this same purpose. The `nn` package defines a set of **Modules**, which are roughly equivalent to neural network layers. A Module receives input Variables and computes output Variables, but may also hold internal state such as Variables containing learnable parameters. The `nn` package also defines a set of useful loss functions that are commonly used when training neural networks.

In this example we use the `nn` package to implement our two-layer network:

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Variables for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Variable of input data to the Module and it produces
    # a Variable of output data.
    y_pred = model(x)

    # Compute and print loss. We pass Variables containing the predicted and true
    # values of y, and the loss function returns a Variable containing the
    # loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Variables with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Variable, so
```

```
    # we can access its data and gradients like we did before.
    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

PyTorch: optim

Up to this point we have updated the weights of our models by manually mutating the `.data` member for Variables holding learnable parameters. This is not a huge burden for simple optimization algorithms like stochastic gradient descent, but in practice we often train neural networks using more sophisticated optimizers like AdaGrad, RMSProp, Adam, etc.

The `optim` package in PyTorch abstracts the idea of an optimization algorithm and provides implementations of commonly used optimization algorithms.

In this example we will use the `nn` package to define our model as before, but we will optimize the model using the Adam algorithm provided by the `optim` package:

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(size_average=False)

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use Adam; the optim package contains many other
# optimization algoriths. The first argument to the Adam constructor tells the
# optimizer which Variables it should update.
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()
```

```
    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

PyTorch: Custom nn Modules

Sometimes you will want to specify models that are more complex than a sequence of existing Modules; for these cases you can define your own Modules by subclassing `nn.Module` and defining a `forward` which receives input Variables and produces output Variables using other modules or other autograd operations on Variables.

In this example we implement our two-layer network as a custom Module subclass:

```
# -*- coding: utf-8 -*-
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
```

```
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

PyTorch: Control Flow + Weight Sharing

As an example of dynamic graphs and weight sharing, we implement a very strange model: a fully-connected ReLU network that on each forward pass chooses a random number between 1 and 4 and uses that many hidden layers, reusing the same weights multiple times to compute the innermost hidden layers.

For this model we can use normal Python flow control to implement the loop, and we can implement weight sharing among the innermost layers by simply reusing the same Module multiple times when defining the forward pass.

We can easily implement this model as a Module subclass:

```
# -*- coding: utf-8 -*-
import random
import torch
from torch.autograd import Variable


class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we construct three nn.Linear instances that we will use
        in the forward pass.
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 0, 1, 2, or 3
        and reuse the middle_linear Module that many times to compute hidden layer
        representations.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same Module many
        times when defining a computational graph. This is a big improvement from Lua
        Torch, where each Module could be used only once.
        """
```

```
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

1.3.4 Examples

You can browse the above examples here.

Tensors

Warm-up: numpy

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x using Euclidean error.

This implementation uses numpy to manually compute the forward pass, loss, and backward pass.

A numpy array is a generic n-dimensional array; it does not know anything about deep learning or gradients or computational graphs, and is just a way to perform generic numeric computations.

```
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
```

```
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Total running time of the script: (0 minutes 0.000 seconds)

PyTorch: Tensors

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses PyTorch tensors to manually compute the forward pass, loss, and backward pass.

A PyTorch Tensor is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation.

The biggest difference between a numpy array and a PyTorch Tensor is that a PyTorch Tensor can run on either CPU or GPU. To run operations on the GPU, just cast the Tensor to a cuda datatype.

```
import torch


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Total running time of the script: (0 minutes 0.000 seconds)

Autograd

PyTorch: Variables and autograd

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation computes the forward pass using operations on PyTorch Variables, and uses PyTorch autograd to compute gradients.

A PyTorch Variable is a wrapper around a PyTorch Tensor, and represents a node in a computational graph. If x is a Variable then x.data is a Tensor giving its value, and x.grad is another Variable holding the gradient of x with respect to some scalar value.

PyTorch Variables have the same API as PyTorch tensors: (almost) any operation you can do on a Tensor you can also do on a Variable; the difference is that autograd allows you to automatically compute gradients.

```
import torch
from torch.autograd import Variable

dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU
```

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs, and wrap them in Variables.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Variables during the backward pass.
x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

# Create random Tensors for weights, and wrap them in Variables.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Variables during the backward pass.
w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y using operations on Variables; these
    # are exactly the same operations we used to compute the forward pass using
    # Tensors, but we do not need to keep references to intermediate values since
    # we are not implementing the backward pass by hand.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Compute and print loss using operations on Variables.
    # Now loss is a Variable of shape (1,) and loss.data is a Tensor of shape
    # (1,); loss.data[0] is a scalar value holding the loss.
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.data[0])

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Variables with requires_grad=True.
    # After this call w1.grad and w2.grad will be Variables holding the gradient
    # of the loss with respect to w1 and w2 respectively.
    loss.backward()

    # Update weights using gradient descent; w1.data and w2.data are Tensors,
    # w1.grad and w2.grad are Variables and w1.grad.data and w2.grad.data are
    # Tensors.
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data

    # Manually zero the gradients after updating weights
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

Total running time of the script: (0 minutes 0.000 seconds)

PyTorch: Defining new autograd functions

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation computes the forward pass using operations on PyTorch Variables, and uses PyTorch autograd to compute gradients.

In this implementation we implement our own custom autograd function to perform the ReLU function.

```
import torch
from torch.autograd import Variable


class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in).type(dtype), requires_grad=False)
y = Variable(torch.randn(N, D_out).type(dtype), requires_grad=False)

# Create random Tensors for weights, and wrap them in Variables.
w1 = Variable(torch.randn(D_in, H).type(dtype), requires_grad=True)
w2 = Variable(torch.randn(H, D_out).type(dtype), requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply

    # Forward pass: compute predicted y using operations on Variables; we compute
    # ReLU using our custom autograd operation.
    y_pred = relu(x.mm(w1)).mm(w2)
```

```
    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.data[0])

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    w1.data -= learning_rate * w1.grad.data
    w2.data -= learning_rate * w2.grad.data

    # Manually zero the gradients after updating weights
    w1.grad.data.zero_()
    w2.grad.data.zero_()
```

Total running time of the script: (0 minutes 0.000 seconds)

TensorFlow: Static Graphs

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses basic TensorFlow operations to set up a computational graph, then executes the graph many times to actually train the network.

One of the main differences between TensorFlow and PyTorch is that TensorFlow uses static computational graphs while PyTorch uses dynamic computational graphs.

In TensorFlow we first set up the computational graph, then execute the same graph many times.

```
import tensorflow as tf
import numpy as np

# First we set up the computational graph:

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create placeholders for the input and target data; these will be filled
# with real data when we execute the graph.
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# Create Variables for the weights and initialize them with random data.
# A TensorFlow Variable persists its value across executions of the graph.
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# Forward pass: Compute the predicted y using operations on TensorFlow Tensors.
# Note that this code does not actually perform any numeric operations; it
# merely sets up the computational graph that we will later execute.
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)
```

```
# Compute loss using operations on TensorFlow Tensors
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# Compute gradient of the loss with respect to w1 and w2.
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

# Update the weights using gradient descent. To actually update the weights
# we need to evaluate new_w1 and new_w2 when executing the graph. Note that
# in TensorFlow the the act of updating the value of the weights is part of
# the computational graph; in PyTorch this happens outside the computational
# graph.
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

# Now we have built our computational graph, so we enter a TensorFlow session to
# actually execute the graph.
with tf.Session() as sess:
    # Run the graph once to initialize the Variables w1 and w2.
    sess.run(tf.global_variables_initializer())

    # Create numpy arrays holding the actual data for the inputs x and targets
    # y
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)
    for _ in range(500):
        # Execute the graph many times. Each time it executes we want to bind
        # x_value to x and y_value to y, specified with the feed_dict argument.
        # Each time we execute the graph we want to compute the values for loss,
        # new_w1, and new_w2; the values of these Tensors are returned as numpy
        # arrays.
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)
```

Total running time of the script: (0 minutes 0.000 seconds)

nn module

PyTorch: nn

A fully-connected ReLU network with one hidden layer, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses the nn package from PyTorch to build the network. PyTorch autograd makes it easy to define computational graphs and take gradients, but raw autograd can be a bit too low-level for defining complex neural networks; this is where the nn package can help. The nn package defines a set of Modules, which you can think of as a neural network layer that has produces output from input and may have some trainable weights.

```
import torch
from torch.autograd import Variable

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10
```

```
# Create random Tensors to hold inputs and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Variables for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(size_average=False)

learning_rate = 1e-4
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Variable of input data to the Module and it produces
    # a Variable of output data.
    y_pred = model(x)

    # Compute and print loss. We pass Variables containing the predicted and true
    # values of y, and the loss function returns a Variable containing the
    # loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Variables with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Variable, so
    # we can access its data and gradients like we did before.
    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Total running time of the script: (0 minutes 0.000 seconds)

PyTorch: optim

A fully-connected ReLU network with one hidden layer, trained to predict y from x by minimizing squared Euclidean distance.

This implementation uses the nn package from PyTorch to build the network.

Rather than manually updating the weights of the model as we have been doing, we use the optim package to define an Optimizer that will update the weights for us. The optim package defines many optimization algorithms that are

commonly used for deep learning, including SGD+momentum, RMSProp, Adam, etc.

```
import torch
from torch.autograd import Variable

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables.
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(size_average=False)

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use Adam; the optim package contains many other
# optimization algoriths. The first argument to the Adam constructor tells the
# optimizer which Variables it should update.
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    print(t, loss.data[0])

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

Total running time of the script: (0 minutes 0.000 seconds)

PyTorch: Custom nn Modules

A fully-connected ReLU network with one hidden layer, trained to predict y from x by minimizing squared Euclidean distance.

This implementation defines the model as a custom Module subclass. Whenever you want a model more complex than

a simple sequence of existing Modules you will need to define your model this way.

```
import torch
from torch.autograd import Variable


class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we instantiate two nn.Linear modules and assign them as
        member variables.
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        In the forward function we accept a Variable of input data and we must return
        a Variable of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Variables.
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Total running time of the script: (0 minutes 0.000 seconds)

PyTorch: Control Flow + Weight Sharing

To showcase the power of PyTorch dynamic graphs, we will implement a very strange model: a fully-connected ReLU network that on each forward pass randomly chooses a number between 1 and 4 and has that many hidden layers, reusing the same weights multiple times to compute the innermost hidden layers.

```
import random
import torch
from torch.autograd import Variable


class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we construct three nn.Linear instances that we will use
        in the forward pass.
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 0, 1, 2, or 3
        and reuse the middle_linear Module that many times to compute hidden layer
        representations.

        Since each forward pass builds a dynamic computation graph, we can use normal
        Python control-flow operators like loops or conditional statements when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same Module many
        times when defining a computational graph. This is a big improvement from Lua
        Torch, where each Module could be used only once.
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. Training this strange model with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
```

```
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Total running time of the script: (0 minutes 0.000 seconds)

1.4 Transfer Learning tutorial

Author: Sasank Chilamkurthy

In this tutorial, you will learn how to train your network using transfer learning. You can read more about the transfer learning at cs231n notes

Quoting this notes,

> In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

These two major transfer learning scenarios looks as follows:

- **Finetuning the convnet**: Instead of random initializaion, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.
- **ConvNet as fixed feature extractor**: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

```
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from torch.autograd import Variable
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy

plt.ion()   # interactive mode
```

1.4.1 Load Data

We will use torchvision and torch.utils.data packages for loading the data.

The problem we're going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.

This dataset is a very small subset of imagenet.

Note: Download the data from here and extract it to the current directory.

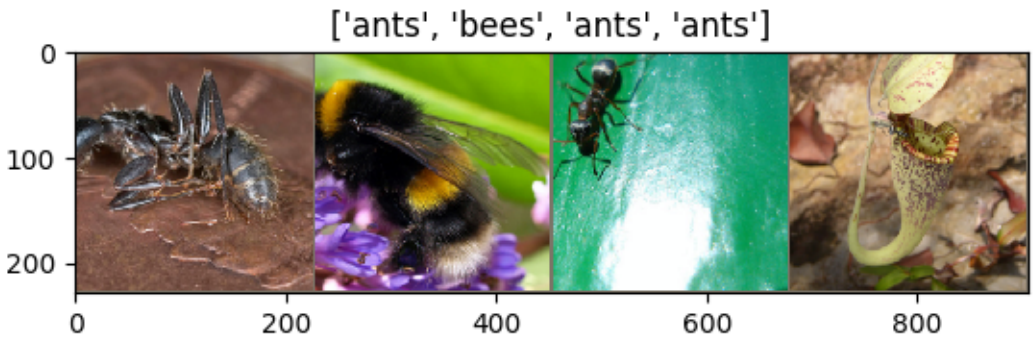
```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Scale(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = 'hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

use_gpu = torch.cuda.is_available()
```

Visualize a few images

Let's visualize a few training images so as to understand the data augmentations.

```
def imshow(inp, title=None):
    """Imshow for Tensor."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
```

```
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])
```



1.4.2 Training the model

Now, let's write a general function to train a model. Here, we will illustrate:

- Scheduling the learning rate
- Saving the best model

In the following, parameter `scheduler` is an LR scheduler object from `torch.optim.lr_scheduler`.

```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
```

```
    best_acc = 0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train(True)  # Set model to training mode
            else:
                model.train(False)  # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for data in dataloaders[phase]:
                # get the inputs
                inputs, labels = data

                # wrap them in Variable
                if use_gpu:
                    inputs = Variable(inputs.cuda())
                    labels = Variable(labels.cuda())
                else:
                    inputs, labels = Variable(inputs), Variable(labels)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                outputs = model(inputs)
                _, preds = torch.max(outputs.data, 1)
                loss = criterion(outputs, labels)

                # backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

                # statistics
                running_loss += loss.data[0] * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()
```

```
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

Visualizing the model predictions

Generic function to display predictions for a few images

```
def visualize_model(model, num_images=6):
    images_so_far = 0
    fig = plt.figure()

    for i, data in enumerate(dataloaders['val']):
        inputs, labels = data
        if use_gpu:
            inputs, labels = Variable(inputs.cuda()), Variable(labels.cuda())
        else:
            inputs, labels = Variable(inputs), Variable(labels)

        outputs = model(inputs)
        _, preds = torch.max(outputs.data, 1)

        for j in range(inputs.size()[0]):
            images_so_far += 1
            ax = plt.subplot(num_images//2, 2, images_so_far)
            ax.axis('off')
            ax.set_title('predicted: {}'.format(class_names[preds[j]]))
            imshow(inputs.cpu().data[j])

            if images_so_far == num_images:
                return
```

1.4.3 Finetuning the convnet

Load a pretrained model and reset final fully connected layer.

```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)

if use_gpu:
    model_ft = model_ft.cuda()

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
```

```
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

Train and evaluate

It should take around 15-25 min on CPU. On GPU though, it takes less than a minute.

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=25)
```

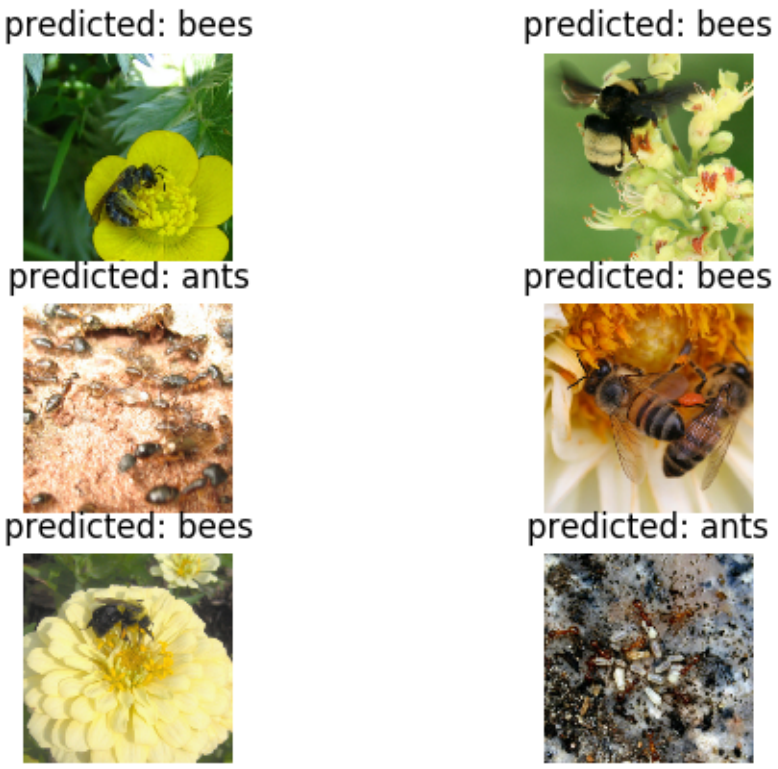
Out:

```
Epoch 0/24
----------
train Loss: 0.5510 Acc: 0.7049
val Loss: 0.2660 Acc: 0.9150

Epoch 1/24
----------
train Loss: 0.6444 Acc: 0.7295
val Loss: 0.2219 Acc: 0.9150

Epoch 2/24
----------
train Loss: 0.5623 Acc: 0.7418
val Loss: 0.2673 Acc: 0.9281

Epoch 3/24
----------
train Loss: 0.3194 Acc: 0.8607
val Loss: 0.2687 Acc: 0.9020

Epoch 4/24
----------
train Loss: 0.5327 Acc: 0.7828
val Loss: 0.4407 Acc: 0.8562

Epoch 5/24
----------
train Loss: 0.5856 Acc: 0.7705
val Loss: 0.2788 Acc: 0.9085

Epoch 6/24
----------
train Loss: 0.4434 Acc: 0.8115
val Loss: 0.2683 Acc: 0.9020

Epoch 7/24
----------
train Loss: 0.3366 Acc: 0.8443
val Loss: 0.2581 Acc: 0.9216

Epoch 8/24
----------
train Loss: 0.2412 Acc: 0.9180
val Loss: 0.2124 Acc: 0.9216
```

```
Epoch 9/24
----------
train Loss: 0.3449 Acc: 0.8361
val Loss: 0.2084 Acc: 0.9216

Epoch 10/24
----------
train Loss: 0.3393 Acc: 0.8607
val Loss: 0.2096 Acc: 0.9346

Epoch 11/24
----------
train Loss: 0.2633 Acc: 0.9180
val Loss: 0.2029 Acc: 0.9346

Epoch 12/24
----------
train Loss: 0.2605 Acc: 0.8852
val Loss: 0.2193 Acc: 0.9216

Epoch 13/24
----------
train Loss: 0.2383 Acc: 0.8934
val Loss: 0.2245 Acc: 0.9150

Epoch 14/24
----------
train Loss: 0.2223 Acc: 0.9057
val Loss: 0.2165 Acc: 0.9346

Epoch 15/24
----------
train Loss: 0.2860 Acc: 0.8648
val Loss: 0.2059 Acc: 0.9346

Epoch 16/24
----------
train Loss: 0.2622 Acc: 0.8730
val Loss: 0.2099 Acc: 0.9281

Epoch 17/24
----------
train Loss: 0.2661 Acc: 0.8852
val Loss: 0.2234 Acc: 0.9346

Epoch 18/24
----------
train Loss: 0.2601 Acc: 0.8730
val Loss: 0.2500 Acc: 0.9085

Epoch 19/24
----------
train Loss: 0.2932 Acc: 0.8893
val Loss: 0.2049 Acc: 0.9346

Epoch 20/24
----------
```

```
train Loss: 0.2807 Acc: 0.8730
val Loss: 0.2052 Acc: 0.9346

Epoch 21/24
----------
train Loss: 0.2718 Acc: 0.8770
val Loss: 0.2033 Acc: 0.9412

Epoch 22/24
----------
train Loss: 0.2572 Acc: 0.8852
val Loss: 0.2009 Acc: 0.9412

Epoch 23/24
----------
train Loss: 0.2310 Acc: 0.9016
val Loss: 0.2096 Acc: 0.9281

Epoch 24/24
----------
train Loss: 0.3295 Acc: 0.8361
val Loss: 0.2041 Acc: 0.9281

Training complete in 46m 44s
Best val Acc: 0.941176
```

```
visualize_model(model_ft)
```



1.4.4 ConvNet as fixed feature extractor

Here, we need to freeze all the network except the final layer. We need to set `requires_grad == False` to freeze the parameters so that the gradients are not computed in `backward()`.

You can read more about this in the documentation here.

```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

if use_gpu:
    model_conv = model_conv.cuda()

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opoosed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
```

```
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

Train and evaluate

On CPU this will take about half the time compared to previous scenario. This is expected as gradients don't need to be computed for most of the network. However, forward does need to be computed.

```
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=25)
```

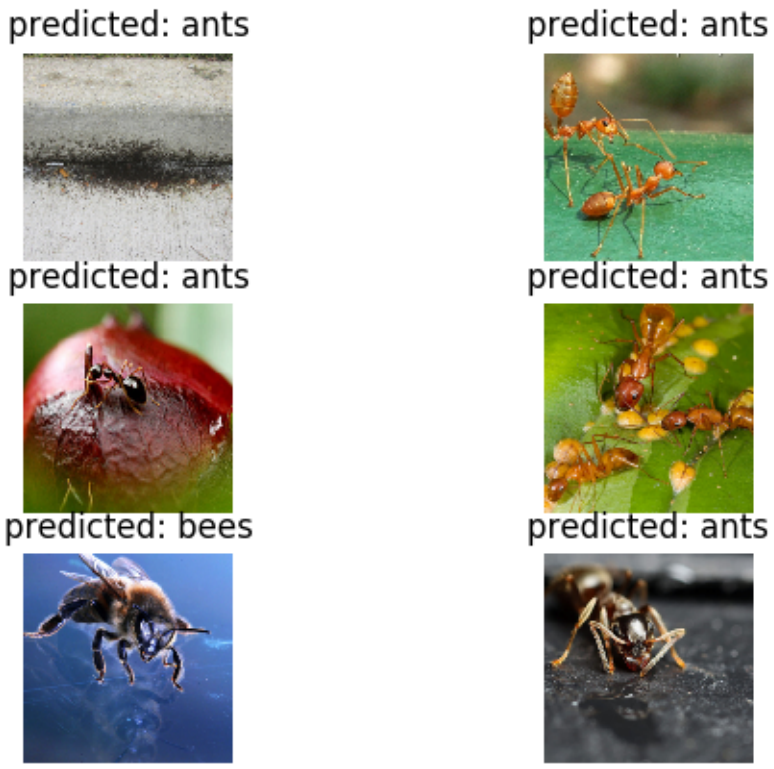
Out:

```
Epoch 0/24
----------
train Loss: 0.6092 Acc: 0.6598
val Loss: 0.3315 Acc: 0.8693

Epoch 1/24
----------
train Loss: 0.6091 Acc: 0.7459
val Loss: 0.1904 Acc: 0.9281

Epoch 2/24
----------
train Loss: 0.4684 Acc: 0.7746
val Loss: 0.2894 Acc: 0.8889

Epoch 3/24
----------
train Loss: 0.4734 Acc: 0.7746
val Loss: 0.2644 Acc: 0.9216

Epoch 4/24
----------
train Loss: 0.4293 Acc: 0.8156
val Loss: 0.2668 Acc: 0.9150

Epoch 5/24
----------
train Loss: 0.5169 Acc: 0.7787
val Loss: 0.2560 Acc: 0.9281

Epoch 6/24
----------
train Loss: 0.4968 Acc: 0.7992
val Loss: 0.3830 Acc: 0.8693

Epoch 7/24
----------
train Loss: 0.3388 Acc: 0.8197
val Loss: 0.2048 Acc: 0.9216

Epoch 8/24
----------
train Loss: 0.4599 Acc: 0.8279
val Loss: 0.2077 Acc: 0.9346
```

```
Epoch 9/24
----------
train Loss: 0.3965 Acc: 0.8402
val Loss: 0.3037 Acc: 0.9085

Epoch 10/24
----------
train Loss: 0.3463 Acc: 0.8607
val Loss: 0.2196 Acc: 0.9346

Epoch 11/24
----------
train Loss: 0.3813 Acc: 0.8074
val Loss: 0.2091 Acc: 0.9281

Epoch 12/24
----------
train Loss: 0.3461 Acc: 0.8566
val Loss: 0.2127 Acc: 0.9216

Epoch 13/24
----------
train Loss: 0.3150 Acc: 0.8648
val Loss: 0.2048 Acc: 0.9150

Epoch 14/24
----------
train Loss: 0.2873 Acc: 0.8811
val Loss: 0.2136 Acc: 0.9412

Epoch 15/24
----------
train Loss: 0.3522 Acc: 0.8566
val Loss: 0.2071 Acc: 0.9150

Epoch 16/24
----------
train Loss: 0.4036 Acc: 0.8033
val Loss: 0.2186 Acc: 0.9150

Epoch 17/24
----------
train Loss: 0.3014 Acc: 0.8607
val Loss: 0.2184 Acc: 0.9216

Epoch 18/24
----------
train Loss: 0.3336 Acc: 0.8361
val Loss: 0.2116 Acc: 0.9346

Epoch 19/24
----------
train Loss: 0.3256 Acc: 0.8525
val Loss: 0.2245 Acc: 0.9346

Epoch 20/24
----------
```

```
train Loss: 0.2924 Acc: 0.8689
val Loss: 0.2161 Acc: 0.9346

Epoch 21/24
----------
train Loss: 0.3236 Acc: 0.8689
val Loss: 0.2320 Acc: 0.9346

Epoch 22/24
----------
train Loss: 0.3187 Acc: 0.8484
val Loss: 0.2291 Acc: 0.9346

Epoch 23/24
----------
train Loss: 0.2911 Acc: 0.8566
val Loss: 0.2290 Acc: 0.9281

Epoch 24/24
----------
train Loss: 0.2715 Acc: 0.8811
val Loss: 0.2069 Acc: 0.9281

Training complete in 21m 58s
Best val Acc: 0.941176
```

```
visualize_model(model_conv)

plt.ioff()
plt.show()
```



Total running time of the script: (69 minutes 8.949 seconds)

`Download Python source code: transfer_learning_tutorial.py`

`Download Jupyter notebook: transfer_learning_tutorial.ipynb`

Generated by Sphinx-Gallery

1.5 Data Loading and Processing Tutorial

Author: Sasank Chilamkurthy

A lot of effort in solving any machine learning problem goes in to preparing the data. PyTorch provides many tools to make data loading easy and hopefully, to make your code more readable. In this tutorial, we will see how to load and preprocess/augment data from a non trivial dataset.

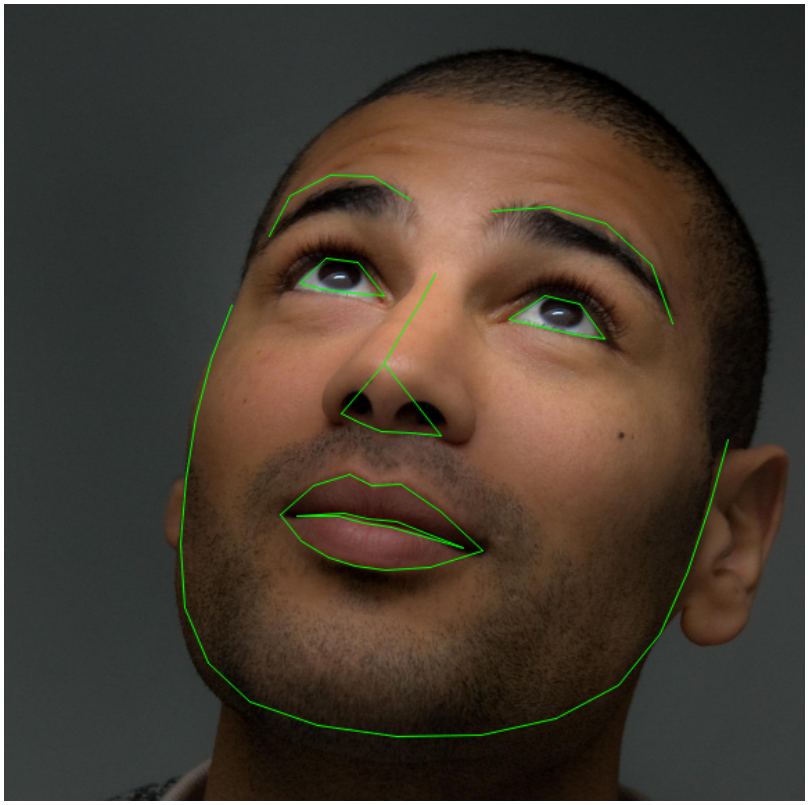
To run this tutorial, please make sure the following packages are installed:

- `scikit-image`: For image io and transforms
- `pandas`: For easier csv parsing

```
from __future__ import print_function, division
import os
import torch
import pandas as pd
from skimage import io, transform
```

```
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils

# Ignore warnings
import warnings
warnings.filterwarnings("ignore")

plt.ion()    # interactive mode
```

The dataset we are going to deal with is that of facial pose. This means that a face is annotated like this:



Over all, 68 different landmark points are annotated for each face.

Note: Download the dataset from here so that the images are in a directory named 'faces/'. This dataset was actually generated by applying excellent dlib's pose estimation on a few images from imagenet tagged as 'face'.

Dataset comes with a csv file with annotations which looks like this:

```
image_name,part_0_x,part_0_y,part_1_x,part_1_y,part_2_x, ... ,part_67_x,part_67_y
0805personali01.jpg,27,83,27,98, ... 84,134
1084239450_e76e00b7e7.jpg,70,236,71,257, ... ,128,312
```

Let's quickly read the CSV and get the annotations in an (N, 2) array where N is the number of landmarks.

```
landmarks_frame = pd.read_csv('faces/face_landmarks.csv')
```

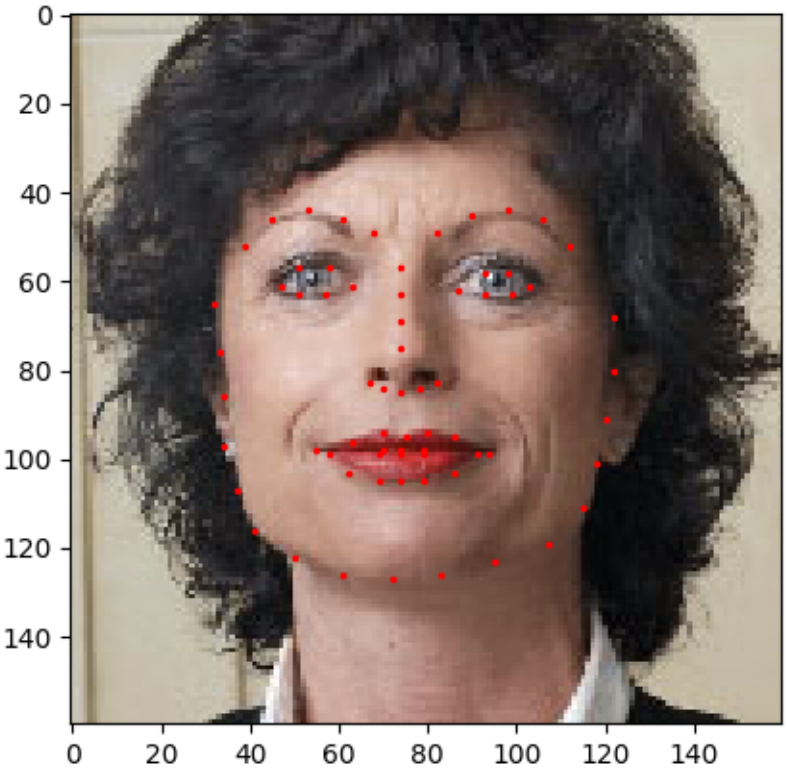
```
n = 65
img_name = landmarks_frame.iloc[n, 0]
landmarks = landmarks_frame.iloc[n, 1:].as_matrix()
landmarks = landmarks.astype('float').reshape(-1, 2)

print('Image name: {}'.format(img_name))
print('Landmarks shape: {}'.format(landmarks.shape))
print('First 4 Landmarks: {}'.format(landmarks[:4]))
```

Out:

```
Image name: person-7.jpg
Landmarks shape: (68, 2)
First 4 Landmarks: [[ 32.  65.]
 [ 33.  76.]
 [ 34.  86.]
 [ 34.  97.]]
```

Let's write a simple helper function to show an image and its landmarks and use it to show a sample.

```
def show_landmarks(image, landmarks):
    """Show image with landmarks"""
    plt.imshow(image)
    plt.scatter(landmarks[:, 0], landmarks[:, 1], s=10, marker='.', c='r')
    plt.pause(0.001)  # pause a bit so that plots are updated

plt.figure()
show_landmarks(io.imread(os.path.join('faces/', img_name)),
               landmarks)
plt.show()
```



1.5.1 Dataset class

`torch.utils.data.Dataset` is an abstract class representing a dataset. Your custom dataset should inherit `Dataset` and override the following methods:

- `__len__` so that `len(dataset)` returns the size of the dataset.
- `__getitem__` to support the indexing such that `dataset[i]` can be used to get ith sample

Let's create a dataset class for our face landmarks dataset. We will read the csv in `__init__` but leave the reading of images to `__getitem__`. This is memory efficient because all the images are not stored in the memory at once but read as required.

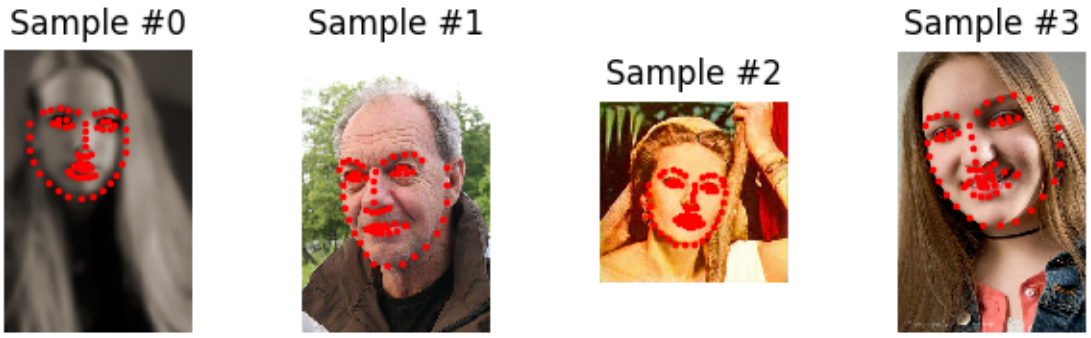
Sample of our dataset will be a dict `{'image': image, 'landmarks': landmarks}`. Our datset will take an optional argument `transform` so that any required processing can be applied on the sample. We will see the usefulness of `transform` in the next section.

```
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
```

```
                on a sample.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.landmarks_frame)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir,
                                self.landmarks_frame.iloc[idx, 0])
        image = io.imread(img_name)
        landmarks = self.landmarks_frame.iloc[idx, 1:].as_matrix()
        landmarks = landmarks.astype('float').reshape(-1, 2)
        sample = {'image': image, 'landmarks': landmarks}

        if self.transform:
            sample = self.transform(sample)

        return sample
```

Let's instantiate this class and iterate through the data samples. We will print the sizes of first 4 samples and show their landmarks.

```
face_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                    root_dir='faces/')

fig = plt.figure()

for i in range(len(face_dataset)):
    sample = face_dataset[i]

    print(i, sample['image'].shape, sample['landmarks'].shape)

    ax = plt.subplot(1, 4, i + 1)
    plt.tight_layout()
    ax.set_title('Sample #{}'.format(i))
    ax.axis('off')
    show_landmarks(**sample)

    if i == 3:
        plt.show()
        break
```



Out:

```
0 (324, 215, 3) (68, 2)
1 (500, 333, 3) (68, 2)
2 (250, 258, 3) (68, 2)
3 (434, 290, 3) (68, 2)
```

1.5.2 Transforms

One issue we can see from the above is that the samples are not of the same size. Most neural networks expect the images of a fixed size. Therefore, we will need to write some prepocessing code. Let's create three transforms:

- `Rescale`: to scale the image
- `RandomCrop`: to crop from image randomly. This is data augmentation.
- `ToTensor`: to convert the numpy images to torch images (we need to swap axes).

We will write them as callable classes instead of simple functions so that parameters of the transform need not be passed everytime it's called. For this, we just need to implement `__call__` method and if required, `__init__` method. We can then use a transform like this:

```
tsfm = Transform(params)
transformed_sample = tsfm(sample)
```

Observe below how these transforms had to be applied both on the image and landmarks.

```
class Rescale(object):
    """Rescale the image in a sample to a given size.

    Args:
        output_size (tuple or int): Desired output size. If tuple, output is
            matched to output_size. If int, smaller of image edges is matched
            to output_size keeping aspect ratio the same.
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        if isinstance(self.output_size, int):
            if h > w:
                new_h, new_w = self.output_size * h / w, self.output_size
            else:
                new_h, new_w = self.output_size, self.output_size * w / h
        else:
            new_h, new_w = self.output_size

        new_h, new_w = int(new_h), int(new_w)

        img = transform.resize(image, (new_h, new_w))

        # h and w are swapped for landmarks because for images,
        # x and y axes are axis 1 and 0 respectively
        landmarks = landmarks * [new_w / w, new_h / h]

        return {'image': img, 'landmarks': landmarks}


class RandomCrop(object):
    """Crop randomly the image in a sample.

    Args:
        output_size (tuple or int): Desired output size. If int, square crop
            is made.
    """

    def __init__(self, output_size):
        assert isinstance(output_size, (int, tuple))
        if isinstance(output_size, int):
            self.output_size = (output_size, output_size)
        else:
            assert len(output_size) == 2
            self.output_size = output_size

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        h, w = image.shape[:2]
        new_h, new_w = self.output_size
```

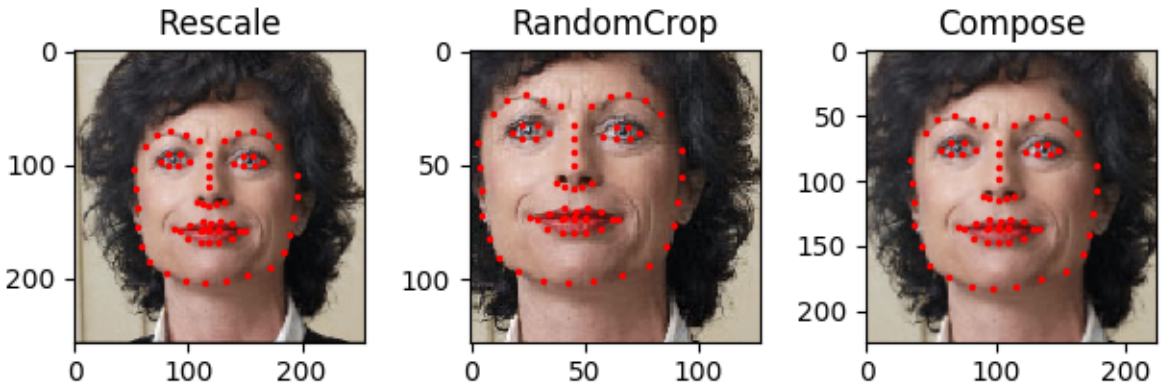
```
        top = np.random.randint(0, h - new_h)
        left = np.random.randint(0, w - new_w)

        image = image[top: top + new_h,
                      left: left + new_w]

        landmarks = landmarks - [left, top]

        return {'image': image, 'landmarks': landmarks}


class ToTensor(object):
    """Convert ndarrays in sample to Tensors."""

    def __call__(self, sample):
        image, landmarks = sample['image'], sample['landmarks']

        # swap color axis because
        # numpy image: H x W x C
        # torch image: C X H X W
        image = image.transpose((2, 0, 1))
        return {'image': torch.from_numpy(image),
                'landmarks': torch.from_numpy(landmarks)}
```

Compose transforms

Now, we apply the transforms on an sample.

Let's say we want to rescale the shorter side of the image to 256 and then randomly crop a square of size 224 from it. i.e, we want to compose `Rescale` and `RandomCrop` transforms. `torchvision.transforms.Compose` is a simple callable class which allows us to do this.

```
scale = Rescale(256)
crop = RandomCrop(128)
composed = transforms.Compose([Rescale(256),
                               RandomCrop(224)])

# Apply each of the above transforms on sample.
fig = plt.figure()
sample = face_dataset[65]
for i, tsfrm in enumerate([scale, crop, composed]):
    transformed_sample = tsfrm(sample)

    ax = plt.subplot(1, 3, i + 1)
    plt.tight_layout()
    ax.set_title(type(tsfrm).__name__)
    show_landmarks(**transformed_sample)

plt.show()
```



1.5.3 Iterating through the dataset

Let's put this all together to create a dataset with composed transforms. To summarize, every time this dataset is sampled:

- An image is read from the file on the fly
- Transforms are applied on the read image
- Since one of the transforms is random, data is augmentated on sampling

We can iterate over the created dataset with a `for i in range` loop as before.

```
transformed_dataset = FaceLandmarksDataset(csv_file='faces/face_landmarks.csv',
                                           root_dir='faces/',
                                           transform=transforms.Compose([
                                               Rescale(256),
                                               RandomCrop(224),
                                               ToTensor()
                                           ]))

for i in range(len(transformed_dataset)):
    sample = transformed_dataset[i]

    print(i, sample['image'].size(), sample['landmarks'].size())
```

```
    if i == 3:
        break
```

Out:

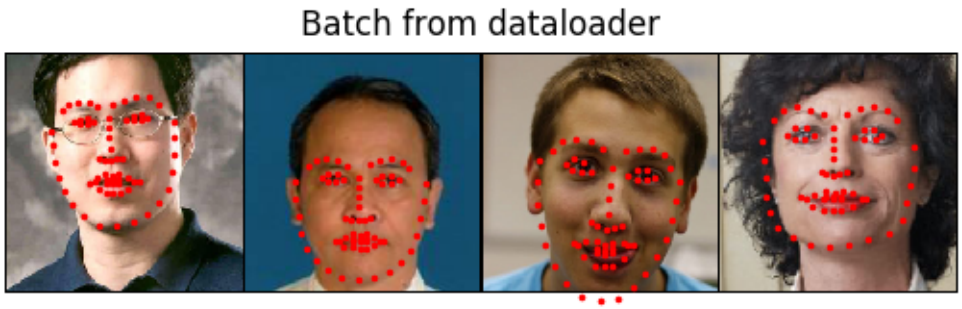
```
0 torch.Size([3, 224, 224]) torch.Size([68, 2])
1 torch.Size([3, 224, 224]) torch.Size([68, 2])
2 torch.Size([3, 224, 224]) torch.Size([68, 2])
3 torch.Size([3, 224, 224]) torch.Size([68, 2])
```

However, we are losing a lot of features by using a simple `for` loop to iterate over the data. In particular, we are missing out on:

- Batching the data
- Shuffling the data
- Load the data in parallel using `multiprocessing` workers.

`torch.utils.data.DataLoader` is an iterator which provides all these features. Parameters used below should be clear. One parameter of interest is `collate_fn`. You can specify how exactly the samples need to be batched using `collate_fn`. However, default collate should work fine for most use cases.

```
dataloader = DataLoader(transformed_dataset, batch_size=4,
                        shuffle=True, num_workers=4)


# Helper function to show a batch
def show_landmarks_batch(sample_batched):
    """Show image with landmarks for a batch of samples."""
    images_batch, landmarks_batch = \
            sample_batched['image'], sample_batched['landmarks']
    batch_size = len(images_batch)
    im_size = images_batch.size(2)

    grid = utils.make_grid(images_batch)
    plt.imshow(grid.numpy().transpose((1, 2, 0)))

    for i in range(batch_size):
        plt.scatter(landmarks_batch[i, :, 0].numpy() + i * im_size,
                    landmarks_batch[i, :, 1].numpy(),
                    s=10, marker='.', c='r')

        plt.title('Batch from dataloader')

for i_batch, sample_batched in enumerate(dataloader):
    print(i_batch, sample_batched['image'].size(),
          sample_batched['landmarks'].size())

    # observe 4th batch and stop.
    if i_batch == 3:
        plt.figure()
        show_landmarks_batch(sample_batched)
        plt.axis('off')
        plt.ioff()
        plt.show()
        break
```



Out:

```
0 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
1 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
2 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
3 torch.Size([4, 3, 224, 224]) torch.Size([4, 68, 2])
```

1.5.4 Afterword: torchvision

In this tutorial, we have seen how to write and use datasets, transforms and dataloader. `torchvision` package provides some common datasets and transforms. You might not even have to write custom classes. One of the more generic datasets available in torchvision is `ImageFolder`. It assumes that images are organized in the following way:

```
root/ants/xxx.png
root/ants/xxy.jpeg
root/ants/xxz.png
.
.
.
root/bees/123.jpg
root/bees/nsdf3.png
root/bees/asd932_.png
```

where 'ants', 'bees' etc. are class labels. Similarly generic transforms which operate on `PIL.Image` like `RandomHorizontalFlip`, `Scale`, are also avaiable. You can use these to write a dataloader like this:

```
import torch
from torchvision import transforms, datasets

data_transform = transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
    ])
hymenoptera_dataset = datasets.ImageFolder(root='hymenoptera_data/train',
                                           transform=data_transform)
dataset_loader = torch.utils.data.DataLoader(hymenoptera_dataset,
                                             batch_size=4, shuffle=True,
                                             num_workers=4)
```

For an example with training code, please see *Transfer Learning tutorial*.

Total running time of the script: (0 minutes 16.090 seconds)

`Download Python source code: data_loading_tutorial.py`

`Download Jupyter notebook: data_loading_tutorial.ipynb`

Generated by Sphinx-Gallery

1.6 Deep Learning for NLP with Pytorch

Author: Robert Guthrie

This tutorial will walk you through the key ideas of deep learning programming using Pytorch. Many of the concepts (such as the computation graph abstraction and autograd) are not unique to Pytorch and are relevant to any deep learning tool kit out there.

I am writing this tutorial to focus specifically on NLP for people who have never written code in any deep learning framework (e.g, TensorFlow, Theano, Keras, Dynet). It assumes working knowledge of core NLP problems: part-of-speech tagging, language modeling, etc. It also assumes familiarity with neural networks at the level of an intro AI class (such as one from the Russel and Norvig book). Usually, these courses cover the basic backpropagation algorithm on feed-forward neural networks, and make the point that they are chains of compositions of linearities and non-linearities. This tutorial aims to get you started writing deep learning code, given you have this prerequisite knowledge.

Note this is about *models*, not data. For all of the models, I just create a few test examples with small dimensionality so you can see how the weights change as it trains. If you have some real data you want to try, you should be able to rip out any of the models from this notebook and use them on it.

1.6.1 Introduction to PyTorch

Introduction to Torch's tensor library

All of deep learning is computations on tensors, which are generalizations of a matrix that can be indexed in more than 2 dimensions. We will see exactly what this means in-depth later. First, lets look what we can do with tensors.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

Creating Tensors

Tensors can be created from Python lists with the torch.Tensor() function.

```
# Create a torch.Tensor object with the given data.  It is a 1D vector
V_data = [1., 2., 3.]
V = torch.Tensor(V_data)
print(V)

# Creates a matrix
M_data = [[1., 2., 3.], [4., 5., 6]]
M = torch.Tensor(M_data)
print(M)

# Create a 3D tensor of size 2x2x2.
T_data = [[[1., 2.], [3., 4.]],
          [[5., 6.], [7., 8.]]]
T = torch.Tensor(T_data)
print(T)
```

Out:

```
1
 2
 3
[torch.FloatTensor of size 3]


 1  2  3
 4  5  6
[torch.FloatTensor of size 2x3]


(0 ,.,.) =
  1  2
  3  4

(1 ,.,.) =
  5  6
  7  8
[torch.FloatTensor of size 2x2x2]
```

What is a 3D tensor anyway? Think about it like this. If you have a vector, indexing into the vector gives you a scalar. If you have a matrix, indexing into the matrix gives you a vector. If you have a 3D tensor, then indexing into the tensor gives you a matrix!

A note on terminology: when I say "tensor" in this tutorial, it refers to any torch.Tensor object. Matrices and vectors are special cases of torch.Tensors, where their dimension is 1 and 2 respectively. When I am talking about 3D tensors, I will explicitly use the term "3D tensor".

```
# Index into V and get a scalar
print(V[0])

# Index into M and get a vector
print(M[0])

# Index into T and get a matrix
print(T[0])
```

Out:

```
1.0

 1
 2
 3
[torch.FloatTensor of size 3]


 1  2
 3  4
[torch.FloatTensor of size 2x2]
```

You can also create tensors of other datatypes. The default, as you can see, is Float. To create a tensor of integer types, try torch.LongTensor(). Check the documentation for more data types, but Float and Long will be the most common.

You can create a tensor with random data and the supplied dimensionality with torch.randn()

```
x = torch.randn((3, 4, 5))
print(x)
```

Out:

```
(0 ,.,.) =
  0.6614  0.2669  0.0617  0.6213 -0.4519
 -0.1661 -1.5228  0.3817 -1.0276 -0.5631
 -0.8923 -0.0583 -0.1955 -0.9656  0.4224
  0.2673 -0.4212 -0.5107 -1.5727 -0.1232

(1 ,.,.) =
  3.5870 -1.8313  1.5987 -1.2770  0.3255
 -0.4791  1.3790  2.5286  0.4107 -0.9880
 -0.9081  0.5423  0.1103 -2.2590  0.6067
 -0.1383  0.8310 -0.2477 -0.8029  0.2366

(2 ,.,.) =
  0.2857  0.6898 -0.6331  0.8795 -0.6842
  0.4533  0.2912 -0.8317 -0.5525  0.6355
 -0.3968 -0.6571 -1.6428  0.9803 -0.0421
 -0.8206  0.3133 -1.1352  0.3773 -0.2824
[torch.FloatTensor of size 3x4x5]
```

Operations with Tensors

You can operate on tensors in the ways you would expect.

```
x = torch.Tensor([1., 2., 3.])
y = torch.Tensor([4., 5., 6.])
z = x + y
print(z)
```

Out:

```
5
 7
 9
[torch.FloatTensor of size 3]
```

See the documentation for a complete list of the massive number of operations available to you. They expand beyond just mathematical operations.

One helpful operation that we will make use of later is concatenation.

```
# By default, it concatenates along the first axis (concatenates rows)
x_1 = torch.randn(2, 5)
y_1 = torch.randn(3, 5)
z_1 = torch.cat([x_1, y_1])
print(z_1)

# Concatenate columns:
x_2 = torch.randn(2, 3)
y_2 = torch.randn(2, 5)
# second arg specifies which axis to concat along
z_2 = torch.cat([x_2, y_2], 1)
print(z_2)

# If your tensors are not compatible, torch will complain.  Uncomment to see the error
# torch.cat([x_1, x_2])
```

Out:

```
-2.5667 -1.4303  0.5009  0.5438 -0.4057
 1.1341 -1.1115  0.3501 -0.7703 -0.1473
 0.6272  1.0935  0.0939  1.2381 -1.3459
 0.5119 -0.6933 -0.1668 -0.9999 -1.6476
 0.8098  0.0554  1.1340 -0.5326  0.6592
[torch.FloatTensor of size 5x5]


-1.5964 -0.3769 -3.1020 -0.0020 -1.0952  0.6016  0.6984 -0.8005
-0.0995 -0.7213  1.2708  1.5381  1.4673  1.5951 -1.5279  1.0156
[torch.FloatTensor of size 2x8]
```

Reshaping Tensors

Use the .view() method to reshape a tensor. This method receives heavy use, because many neural network components expect their inputs to have a certain shape. Often you will need to reshape before passing your data to the component.

```
x = torch.randn(2, 3, 4)
print(x)
print(x.view(2, 12))  # Reshape to 2 rows, 12 columns
# Same as above.  If one of the dimensions is -1, its size can be inferred
print(x.view(2, -1))
```

Out:

```
(0 ,.,.) =
 -0.2020 -1.2865  0.8231 -0.6101
 -1.2960 -0.9434  0.6684  1.1628
 -0.3229  1.8782 -0.5666  0.4016

(1 ,.,.) =
 -0.1153  0.3170  0.5629  0.8662
 -0.3528  0.3482  1.1371 -0.3339
 -1.4724  0.7296 -0.1312 -0.6368
[torch.FloatTensor of size 2x3x4]



Columns 0 to 9
-0.2020 -1.2865  0.8231 -0.6101 -1.2960 -0.9434  0.6684  1.1628 -0.3229  1.8782
-0.1153  0.3170  0.5629  0.8662 -0.3528  0.3482  1.1371 -0.3339 -1.4724  0.7296

Columns 10 to 11
-0.5666  0.4016
-0.1312 -0.6368
[torch.FloatTensor of size 2x12]



Columns 0 to 9
-0.2020 -1.2865  0.8231 -0.6101 -1.2960 -0.9434  0.6684  1.1628 -0.3229  1.8782
-0.1153  0.3170  0.5629  0.8662 -0.3528  0.3482  1.1371 -0.3339 -1.4724  0.7296

Columns 10 to 11
-0.5666  0.4016
-0.1312 -0.6368
[torch.FloatTensor of size 2x12]
```

Computation Graphs and Automatic Differentiation

The concept of a computation graph is essential to efficient deep learning programming, because it allows you to not have to write the back propagation gradients yourself. A computation graph is simply a specification of how your data is combined to give you the output. Since the graph totally specifies what parameters were involved with which operations, it contains enough information to compute derivatives. This probably sounds vague, so lets see what is going on using the fundamental class of Pytorch: autograd.Variable.

First, think from a programmers perspective. What is stored in the torch.Tensor objects we were creating above? Obviously the data and the shape, and maybe a few other things. But when we added two tensors together, we got an output tensor. All this output tensor knows is its data and shape. It has no idea that it was the sum of two other tensors (it could have been read in from a file, it could be the result of some other operation, etc.)

The Variable class keeps track of how it was created. Lets see it in action.

```
# Variables wrap tensor objects
x = autograd.Variable(torch.Tensor([1., 2., 3]), requires_grad=True)
# You can access the data with the .data attribute
print(x.data)

# You can also do all the same operations you did with tensors with Variables.
y = autograd.Variable(torch.Tensor([4., 5., 6]), requires_grad=True)
z = x + y
print(z.data)

# BUT z knows something extra.
print(z.grad_fn)
```

Out:

```
1
 2
 3
[torch.FloatTensor of size 3]


 5
 7
 9
[torch.FloatTensor of size 3]

<AddBackward1 object at 0x10f1ccb70>
```

So Variables know what created them. z knows that it wasn't read in from a file, it wasn't the result of a multiplication or exponential or whatever. And if you keep following z.grad_fn, you will find yourself at x and y.

But how does that help us compute a gradient?

```
# Lets sum up all the entries in z
s = z.sum()
print(s)
print(s.grad_fn)
```

Out:

```
Variable containing:
 21
[torch.FloatTensor of size 1]

<SumBackward0 object at 0x10f1cc9e8>
```

So now, what is the derivative of this sum with respect to the first component of x? In math, we want

$$\frac{\partial s}{\partial x_0}$$

Well, s knows that it was created as a sum of the tensor z. z knows that it was the sum x + y. So

$$s = \overbrace{x_0 + y_0}^{z_0} + \overbrace{x_1 + y_1}^{z_1} + \overbrace{x_2 + y_2}^{z_2}$$

And so s contains enough information to determine that the derivative we want is 1!

Of course this glosses over the challenge of how to actually compute that derivative. The point here is that s is carrying along enough information that it is possible to compute it. In reality, the developers of Pytorch program the sum() and

+ operations to know how to compute their gradients, and run the back propagation algorithm. An in-depth discussion of that algorithm is beyond the scope of this tutorial.

Lets have Pytorch compute the gradient, and see that we were right: (note if you run this block multiple times, the gradient will increment. That is because Pytorch *accumulates* the gradient into the .grad property, since for many models this is very convenient.)

```
# calling .backward() on any variable will run backprop, starting from it.
s.backward()
print(x.grad)
```

Out:

```
Variable containing:
 1
 1
 1
[torch.FloatTensor of size 3]
```

Understanding what is going on in the block below is crucial for being a successful programmer in deep learning.

```
x = torch.randn((2, 2))
y = torch.randn((2, 2))
z = x + y  # These are Tensor types, and backprop would not be possible

var_x = autograd.Variable(x, requires_grad=True)
var_y = autograd.Variable(y, requires_grad=True)
# var_z contains enough information to compute gradients, as we saw above
var_z = var_x + var_y
print(var_z.grad_fn)

var_z_data = var_z.data  # Get the wrapped Tensor object out of var_z...
# Re-wrap the tensor in a new variable
new_var_z = autograd.Variable(var_z_data)

# ... does new_var_z have information to backprop to x and y?
# NO!
print(new_var_z.grad_fn)
# And how could it?  We yanked the tensor out of var_z (that is
# what var_z.data is).  This tensor doesn't know anything about
# how it was computed.  We pass it into new_var_z, and this is all the
# information new_var_z gets.  If var_z_data doesn't know how it was
# computed, theres no way new_var_z will.
# In essence, we have broken the variable away from its past history
```

Out:

```
<AddBackward1 object at 0x10f1cc9e8>
None
```

Here is the basic, extremely important rule for computing with autograd.Variables (note this is more general than Pytorch. There is an equivalent object in every major deep learning toolkit):

If you want the error from your loss function to backpropagate to a component of your network, you MUST NOT break the Variable chain from that component to your loss Variable. If you do, the loss will have no idea your component exists, and its parameters can't be updated.

I say this in bold, because this error can creep up on you in very subtle ways (I will show some such ways below), and it will not cause your code to crash or complain, so you must be careful.

Total running time of the script: (0 minutes 0.003 seconds)

Download Python source code: pytorch_tutorial.py

Download Jupyter notebook: pytorch_tutorial.ipynb

Generated by Sphinx-Gallery

1.6.2 Deep Learning with PyTorch

Deep Learning Building Blocks: Affine maps, non-linearities and objectives

Deep learning consists of composing linearities with non-linearities in clever ways. The introduction of non-linearities allows for powerful models. In this section, we will play with these core components, make up an objective function, and see how the model is trained.

Affine Maps

One of the core workhorses of deep learning is the affine map, which is a function $f(x)$ where

$$f(x) = Ax + b$$

for a matrix A and vectors x, b. The parameters to be learned here are A and b. Often, b is refered to as the *bias* term.

Pytorch and most other deep learning frameworks do things a little differently than traditional linear algebra. It maps the rows of the input instead of the columns. That is, the i'th row of the output below is the mapping of the i'th row of the input under A, plus the bias term. Look at the example below.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

```
lin = nn.Linear(5, 3)  # maps from R^5 to R^3, parameters A, b
# data is 2x5.  A maps from 5 to 3... can we map "data" under A?
data = autograd.Variable(torch.randn(2, 5))
print(lin(data))  # yes
```

Out:

```
Variable containing:
 0.1755 -0.3268 -0.5069
-0.6602  0.2260  0.1089
[torch.FloatTensor of size 2x3]
```

Non-Linearities

First, note the following fact, which will explain why we need non-linearities in the first place. Suppose we have two affine maps $f(x) = Ax + b$ and $g(x) = Cx + d$. What is $f(g(x))$?

$$f(g(x)) = A(Cx + d) + b = ACx + (Ad + b)$$

AC is a matrix and $Ad + b$ is a vector, so we see that composing affine maps gives you an affine map.

From this, you can see that if you wanted your neural network to be long chains of affine compositions, that this adds no new power to your model than just doing a single affine map.

If we introduce non-linearities in between the affine layers, this is no longer the case, and we can build much more powerful models.

There are a few core non-linearities. $\tanh(x), \sigma(x), \text{ReLU}(x)$ are the most common. You are probably wondering: "why these functions? I can think of plenty of other non-linearities." The reason for this is that they have gradients that are easy to compute, and computing gradients is essential for learning. For example

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$$

A quick note: although you may have learned some neural networks in your intro to AI class where $\sigma(x)$ was the default non-linearity, typically people shy away from it in practice. This is because the gradient *vanishes* very quickly as the absolute value of the argument grows. Small gradients means it is hard to learn. Most people default to tanh or ReLU.

```
# In pytorch, most non-linearities are in torch.functional (we have it imported as F)
# Note that non-linearites typically don't have parameters like affine maps do.
# That is, they don't have weights that are updated during training.
data = autograd.Variable(torch.randn(2, 2))
print(data)
print(F.relu(data))
```

Out:

```
Variable containing:
-0.5404 -2.2102
 2.1130 -0.0040
[torch.FloatTensor of size 2x2]

Variable containing:
 0.0000  0.0000
 2.1130  0.0000
[torch.FloatTensor of size 2x2]
```

Softmax and Probabilities

The function $\text{Softmax}(x)$ is also just a non-linearity, but it is special in that it usually is the last operation done in a network. This is because it takes in a vector of real numbers and returns a probability distribution. Its definition is as follows. Let x be a vector of real numbers (positive, negative, whatever, there are no constraints). Then the i'th component of $\text{Softmax}(x)$ is

$$\frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

It should be clear that the output is a probability distribution: each element is non-negative and the sum over all components is 1.

You could also think of it as just applying an element-wise exponentiation operator to the input to make everything non-negative and then dividing by the normalization constant.

```
# Softmax is also in torch.nn.functional
data = autograd.Variable(torch.randn(5))
print(data)
```

```
print(F.softmax(data, dim=0))
print(F.softmax(data, dim=0).sum())  # Sums to 1 because it is a distribution!
print(F.log_softmax(data, dim=0))  # theres also log_softmax
```

Out:

```
Variable containing:
 1.3800
-1.3505
 0.3455
 0.5046
 1.8213
[torch.FloatTensor of size 5]

Variable containing:
 0.2948
 0.0192
 0.1048
 0.1228
 0.4584
[torch.FloatTensor of size 5]

Variable containing:
 1
[torch.FloatTensor of size 1]

Variable containing:
-1.2214
-3.9519
-2.2560
-2.0969
-0.7801
[torch.FloatTensor of size 5]
```

Objective Functions

The objective function is the function that your network is being trained to minimize (in which case it is often called a *loss function* or *cost function*). This proceeds by first choosing a training instance, running it through your neural network, and then computing the loss of the output. The parameters of the model are then updated by taking the derivative of the loss function. Intuitively, if your model is completely confident in its answer, and its answer is wrong, your loss will be high. If it is very confident in its answer, and its answer is correct, the loss will be low.

The idea behind minimizing the loss function on your training examples is that your network will hopefully generalize well and have small loss on unseen examples in your dev set, test set, or in production. An example loss function is the *negative log likelihood loss*, which is a very common objective for multi-class classification. For supervised multi-class classification, this means training the network to minimize the negative log probability of the correct output (or equivalently, maximize the log probability of the correct output).

Optimization and Training

So what we can compute a loss function for an instance? What do we do with that? We saw earlier that autograd.Variable's know how to compute gradients with respect to the things that were used to compute it. Well, since our loss is an autograd.Variable, we can compute gradients with respect to all of the parameters used to compute it! Then we can perform standard gradient updates. Let θ be our parameters, $L(\theta)$ the loss function, and η a positive learning

rate. Then:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta)$$

There are a huge collection of algorithms and active research in attempting to do something more than just this vanilla gradient update. Many attempt to vary the learning rate based on what is happening at train time. You don't need to worry about what specifically these algorithms are doing unless you are really interested. Torch provides many in the torch.optim package, and they are all completely transparent. Using the simplest gradient update is the same as the more complicated algorithms. Trying different update algorithms and different parameters for the update algorithms (like different initial learning rates) is important in optimizing your network's performance. Often, just replacing vanilla SGD with an optimizer like Adam or RMSProp will boost performance noticably.

Creating Network Components in Pytorch

Before we move on to our focus on NLP, lets do an annotated example of building a network in Pytorch using only affine maps and non-linearities. We will also see how to compute a loss function, using Pytorch's built in negative log likelihood, and update parameters by backpropagation.

All network components should inherit from nn.Module and override the forward() method. That is about it, as far as the boilerplate is concerned. Inheriting from nn.Module provides functionality to your component. For example, it makes it keep track of its trainable parameters, you can swap it between CPU and GPU with the .cuda() or .cpu() functions, etc.

Let's write an annotated example of a network that takes in a sparse bag-of-words representation and outputs a probability distribution over two labels: "English" and "Spanish". This model is just logistic regression.

Example: Logistic Regression Bag-of-Words classifier

Our model will map a sparse BOW representation to log probabilities over labels. We assign each word in the vocab an index. For example, say our entire vocab is two words "hello" and "world", with indices 0 and 1 respectively. The BoW vector for the sentence "hello hello hello hello" is

$$[4, 0]$$

For "hello world world hello", it is

$$[2, 2]$$

etc. In general, it is

$$[\text{Count}(\text{hello}), \text{Count}(\text{world})]$$

Denote this BOW vector as x. The output of our network is:

$$\log \text{Softmax}(Ax + b)$$

That is, we pass the input through an affine map and then do log softmax.

```
data = [("me gusta comer en la cafeteria".split(), "SPANISH"),
        ("Give it to me".split(), "ENGLISH"),
        ("No creo que sea una buena idea".split(), "SPANISH"),
        ("No it is not a good idea to get lost at sea".split(), "ENGLISH")]

test_data = [("Yo creo que si".split(), "SPANISH"),
             ("it is lost on me".split(), "ENGLISH")]
```

```
# word_to_ix maps each word in the vocab to a unique integer, which will be its
# index into the Bag of words vector
word_to_ix = {}
for sent, _ in data + test_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)

VOCAB_SIZE = len(word_to_ix)
NUM_LABELS = 2


class BoWClassifier(nn.Module):  # inheriting from nn.Module!

    def __init__(self, num_labels, vocab_size):
        # calls the init function of nn.Module.  Dont get confused by syntax,
        # just always do it in an nn.Module
        super(BoWClassifier, self).__init__()

        # Define the parameters that you will need.  In this case, we need A and b,
        # the parameters of the affine mapping.
        # Torch defines nn.Linear(), which provides the affine map.
        # Make sure you understand why the input dimension is vocab_size
        # and the output is num_labels!
        self.linear = nn.Linear(vocab_size, num_labels)

        # NOTE! The non-linearity log softmax does not have parameters! So we don't␣
↪need
        # to worry about that here

    def forward(self, bow_vec):
        # Pass the input through the linear layer,
        # then pass that through log_softmax.
        # Many non-linearities and other functions are in torch.nn.functional
        return F.log_softmax(self.linear(bow_vec), dim=1)


def make_bow_vector(sentence, word_to_ix):
    vec = torch.zeros(len(word_to_ix))
    for word in sentence:
        vec[word_to_ix[word]] += 1
    return vec.view(1, -1)


def make_target(label, label_to_ix):
    return torch.LongTensor([label_to_ix[label]])


model = BoWClassifier(NUM_LABELS, VOCAB_SIZE)

# the model knows its parameters.  The first output below is A, the second is b.
# Whenever you assign a component to a class variable in the __init__ function
# of a module, which was done with the line
# self.linear = nn.Linear(...)
# Then through some Python magic from the Pytorch devs, your module
# (in this case, BoWClassifier) will store knowledge of the nn.Linear's parameters
```

```
for param in model.parameters():
    print(param)

# To run the model, pass in a BoW vector, but wrapped in an autograd.Variable
sample = data[0]
bow_vector = make_bow_vector(sample[0], word_to_ix)
log_probs = model(autograd.Variable(bow_vector))
print(log_probs)
```

Out:

```
{'me': 0, 'gusta': 1, 'comer': 2, 'en': 3, 'la': 4, 'cafeteria': 5, 'Give': 6, 'it':␣
↪7, 'to': 8, 'No': 9, 'creo': 10, 'que': 11, 'sea': 12, 'una': 13, 'buena': 14, 'idea
↪': 15, 'is': 16, 'not': 17, 'a': 18, 'good': 19, 'get': 20, 'lost': 21, 'at': 22,
↪'Yo': 23, 'si': 24, 'on': 25}
Parameter containing:

Columns 0 to 9
 0.1194  0.0609 -0.1268  0.1274  0.1191  0.1739 -0.1099 -0.0323 -0.0038  0.0286
 0.1152 -0.1136 -0.1743  0.1427 -0.0291  0.1103  0.0630 -0.1471  0.0394  0.0471

Columns 10 to 19
-0.1488 -0.1392  0.1067 -0.0460  0.0958  0.0112  0.0644  0.0431  0.0713  0.0972
-0.1313 -0.0931  0.0669  0.0351 -0.0834 -0.0594  0.1796 -0.0363  0.1106  0.0849

Columns 20 to 25
-0.1816  0.0987 -0.1379 -0.1480  0.0119 -0.0334
-0.1268 -0.1668  0.1882  0.0102  0.1344  0.0406
[torch.FloatTensor of size 2x26]

Parameter containing:
 0.0631
 0.1465
[torch.FloatTensor of size 2]

Variable containing:
-0.5378 -0.8771
[torch.FloatTensor of size 1x2]
```

Which of the above values corresponds to the log probability of ENGLISH, and which to SPANISH? We never defined it, but we need to if we want to train the thing.

```
label_to_ix = {"SPANISH": 0, "ENGLISH": 1}
```

So lets train! To do this, we pass instances through to get log probabilities, compute a loss function, compute the gradient of the loss function, and then update the parameters with a gradient step. Loss functions are provided by Torch in the nn package. nn.NLLLoss() is the negative log likelihood loss we want. It also defines optimization functions in torch.optim. Here, we will just use SGD.

Note that the *input* to NLLLoss is a vector of log probabilities, and a target label. It doesn't compute the log probabilities for us. This is why the last layer of our network is log softmax. The loss function nn.CrossEntropyLoss() is the same as NLLLoss(), except it does the log softmax for you.

```
# Run on test data before we train, just to see a before-and-after
for instance, label in test_data:
    bow_vec = autograd.Variable(make_bow_vector(instance, word_to_ix))
    log_probs = model(bow_vec)
    print(log_probs)
```

```
# Print the matrix column corresponding to "creo"
print(next(model.parameters())[:, word_to_ix["creo"]])

loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Usually you want to pass over the training data several times.
# 100 is much bigger than on a real data set, but real datasets have more than
# two instances.  Usually, somewhere between 5 and 30 epochs is reasonable.
for epoch in range(100):
    for instance, label in data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Make our BOW vector and also we must wrap the target in a
        # Variable as an integer. For example, if the target is SPANISH, then
        # we wrap the integer 0. The loss function then knows that the 0th
        # element of the log probabilities is the log probability
        # corresponding to SPANISH
        bow_vec = autograd.Variable(make_bow_vector(instance, word_to_ix))
        target = autograd.Variable(make_target(label, label_to_ix))

        # Step 3. Run our forward pass.
        log_probs = model(bow_vec)

        # Step 4. Compute the loss, gradients, and update the parameters by
        # calling optimizer.step()
        loss = loss_function(log_probs, target)
        loss.backward()
        optimizer.step()

for instance, label in test_data:
    bow_vec = autograd.Variable(make_bow_vector(instance, word_to_ix))
    log_probs = model(bow_vec)
    print(log_probs)

# Index corresponding to Spanish goes up, English goes down!
print(next(model.parameters())[:, word_to_ix["creo"]])
```

Out:

```
Variable containing:
-0.9297 -0.5020
[torch.FloatTensor of size 1x2]

Variable containing:
-0.6388 -0.7506
[torch.FloatTensor of size 1x2]

Variable containing:
-0.1488
-0.1313
[torch.FloatTensor of size 2]

Variable containing:
-0.2093 -1.6669
```

```
[torch.FloatTensor of size 1x2]

Variable containing:
-2.5330 -0.0828
[torch.FloatTensor of size 1x2]

Variable containing:
 0.2803
-0.5605
[torch.FloatTensor of size 2]
```

We got the right answer! You can see that the log probability for Spanish is much higher in the first example, and the log probability for English is much higher in the second for the test data, as it should be.

Now you see how to make a Pytorch component, pass some data through it and do gradient updates. We are ready to dig deeper into what deep NLP has to offer.

Total running time of the script: (0 minutes 0.094 seconds)

`Download Python source code: deep_learning_tutorial.py`

`Download Jupyter notebook: deep_learning_tutorial.ipynb`

Generated by Sphinx-Gallery

1.6.3 Word Embeddings: Encoding Lexical Semantics

Word embeddings are dense vectors of real numbers, one per word in your vocabulary. In NLP, it is almost always the case that your features are words! But how should you represent a word in a computer? You could store its ascii character representation, but that only tells you what the word *is*, it doesn't say much about what it *means* (you might be able to derive its part of speech from its affixes, or properties from its capitalization, but not much). Even more, in what sense could you combine these representations? We often want dense outputs from our neural networks, where the inputs are $|V|$ dimensional, where V is our vocabulary, but often the outputs are only a few dimensional (if we are only predicting a handful of labels, for instance). How do we get from a massive dimensional space to a smaller dimensional space?

How about instead of ascii representations, we use a one-hot encoding? That is, we represent the word w by

$$\overbrace{[0, 0, \ldots, 1, \ldots, 0, 0]}^{\text{|V| elements}}$$

where the 1 is in a location unique to w. Any other word will have a 1 in some other location, and a 0 everywhere else.

There is an enormous drawback to this representation, besides just how huge it is. It basically treats all words as independent entities with no relation to each other. What we really want is some notion of *similarity* between words. Why? Let's see an example.

Suppose we are building a language model. Suppose we have seen the sentences

- The mathematician ran to the store.
- The physicist ran to the store.
- The mathematician solved the open problem.

in our training data. Now suppose we get a new sentence never before seen in our training data:

- The physicist solved the open problem.

Our language model might do OK on this sentence, but wouldn't it be much better if we could use the following two facts:

- We have seen mathematician and physicist in the same role in a sentence. Somehow they have a semantic relation.
- We have seen mathematician in the same role in this new unseen sentence as we are now seeing physicist.

and then infer that physicist is actually a good fit in the new unseen sentence? This is what we mean by a notion of similarity: we mean *semantic similarity*, not simply having similar orthographic representations. It is a technique to combat the sparsity of linguistic data, by connecting the dots between what we have seen and what we haven't. This example of course relies on a fundamental linguistic assumption: that words appearing in similar contexts are related to each other semantically. This is called the distributional hypothesis.

Getting Dense Word Embeddings

How can we solve this problem? That is, how could we actually encode semantic similarity in words? Maybe we think up some semantic attributes. For example, we see that both mathematicians and physicists can run, so maybe we give these words a high score for the "is able to run" semantic attribute. Think of some other attributes, and imagine what you might score some common words on those attributes.

If each attribute is a dimension, then we might give each word a vector, like this:

$$q_{\text{mathematician}} = \begin{bmatrix} \overbrace{2.3}^{\text{can run}}, & \overbrace{9.4}^{\text{likes coffee}}, & \overbrace{-5.5}^{\text{majored in Physics}}, & \dots \end{bmatrix}$$

$$q_{\text{physicist}} = \begin{bmatrix} \overbrace{2.5}^{\text{can run}}, & \overbrace{9.1}^{\text{likes coffee}}, & \overbrace{6.4}^{\text{majored in Physics}}, & \dots \end{bmatrix}$$

Then we can get a measure of similarity between these words by doing:

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = q_{\text{physicist}} \cdot q_{\text{mathematician}}$$

Although it is more common to normalize by the lengths:

$$\text{Similarity}(\text{physicist}, \text{mathematician}) = \frac{q_{\text{physicist}} \cdot q_{\text{mathematician}}}{\|q\| \|q_{\text{mathematician}}\|} = \cos(\phi)$$

Where ϕ is the angle between the two vectors. That way, extremely similar words (words whose embeddings point in the same direction) will have similarity 1. Extremely dissimilar words should have similarity -1.

You can think of the sparse one-hot vectors from the beginning of this section as a special case of these new vectors we have defined, where each word basically has similarity 0, and we gave each word some unique semantic attribute. These new vectors are *dense*, which is to say their entries are (typically) non-zero.

But these new vectors are a big pain: you could think of thousands of different semantic attributes that might be relevant to determining similarity, and how on earth would you set the values of the different attributes? Central to the idea of deep learning is that the neural network learns representations of the features, rather than requiring the programmer to design them herself. So why not just let the word embeddings be parameters in our model, and then be updated during training? This is exactly what we will do. We will have some *latent semantic attributes* that the network can, in principle, learn. Note that the word embeddings will probably not be interpretable. That is, although with our hand-crafted vectors above we can see that mathematicians and physicists are similar in that they both like coffee, if we allow a neural network to learn the embeddings and see that both mathematicians and physicisits have a large value in the second dimension, it is not clear what that means. They are similar in some latent semantic dimension, but this probably has no interpretation to us.

In summary, **word embeddings are a representation of the *semantics* of a word, efficiently encoding semantic information that might be relevant to the task at hand**. You can embed other things too: part of speech tags, parse trees, anything! The idea of feature embeddings is central to the field.

Word Embeddings in Pytorch

Before we get to a worked example and an exercise, a few quick notes about how to use embeddings in Pytorch and in deep learning programming in general. Similar to how we defined a unique index for each word when making one-hot vectors, we also need to define an index for each word when using embeddings. These will be keys into a lookup table. That is, embeddings are stored as a $|V| \times D$ matrix, where D is the dimensionality of the embeddings, such that the word assigned index i has its embedding stored in the i'th row of the matrix. In all of my code, the mapping from words to indices is a dictionary named word_to_ix.

The module that allows you to use embeddings is torch.nn.Embedding, which takes two arguments: the vocabulary size, and the dimensionality of the embeddings.

To index into this table, you must use torch.LongTensor (since the indices are integers, not floats).

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

```
word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensional embeddings
lookup_tensor = torch.LongTensor([word_to_ix["hello"]])
hello_embed = embeds(autograd.Variable(lookup_tensor))
print(hello_embed)
```

Out:

```
Variable containing:
 0.6614  0.2669  0.0617  0.6213 -0.4519
[torch.FloatTensor of size 1x5]
```

An Example: N-Gram Language Modeling

Recall that in an n-gram language model, given a sequence of words w, we want to compute

$$P(w_i|w_{i-1}, w_{i-2}, \ldots, w_{i-n+1})$$

Where w_i is the ith word of the sequence.

In this example, we will compute the loss function on some training examples and update the parameters with back-propagation.

```
CONTEXT_SIZE = 2
EMBEDDING_DIM = 10
# We will use Shakespeare Sonnet 2
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
```

```
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""".split()
# we should tokenize the input, but we will ignore that for now
# build a list of tuples.  Each tuple is ([ word_i-2, word_i-1 ], target word)
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# print the first 3, just so you can see what they look like
print(trigrams[:3])

vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}


class NGramLanguageModeler(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(NGramLanguageModeler, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs


losses = []
loss_function = nn.NLLLoss()
model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)

for epoch in range(10):
    total_loss = torch.Tensor([0])
    for context, target in trigrams:

        # Step 1. Prepare the inputs to be passed to the model (i.e, turn the words
        # into integer indices and wrap them in variables)
        context_idxs = [word_to_ix[w] for w in context]
        context_var = autograd.Variable(torch.LongTensor(context_idxs))

        # Step 2. Recall that torch *accumulates* gradients. Before passing in a
        # new instance, you need to zero out the gradients from the old
        # instance
        model.zero_grad()

        # Step 3. Run the forward pass, getting log probabilities over next
        # words
        log_probs = model(context_var)

        # Step 4. Compute your loss function. (Again, Torch wants the target
```

```
        # word wrapped in a variable)
        loss = loss_function(log_probs, autograd.Variable(
            torch.LongTensor([word_to_ix[target]])))

        # Step 5. Do the backward pass and update the gradient
        loss.backward()
        optimizer.step()

        total_loss += loss.data
    losses.append(total_loss)
print(losses)  # The loss decreased every iteration over the training data!
```

Out:

```
[(['When', 'forty'], 'winters'), (['forty', 'winters'], 'shall'), (['winters', 'shall
↪'], 'besiege')]
[
 523.6719
[torch.FloatTensor of size 1]
,
 521.1194
[torch.FloatTensor of size 1]
,
 518.5831
[torch.FloatTensor of size 1]
,
 516.0641
[torch.FloatTensor of size 1]
,
 513.5611
[torch.FloatTensor of size 1]
,
 511.0724
[torch.FloatTensor of size 1]
,
 508.5970
[torch.FloatTensor of size 1]
,
 506.1343
[torch.FloatTensor of size 1]
,
 503.6847
[torch.FloatTensor of size 1]
,
 501.2443
[torch.FloatTensor of size 1]
]
```

Exercise: Computing Word Embeddings: Continuous Bag-of-Words

The Continuous Bag-of-Words model (CBOW) is frequently used in NLP deep learning. It is a model that tries to predict words given the context of a few words before and a few words after the target word. This is distinct from language modeling, since CBOW is not sequential and does not have to be probabilistic. Typcially, CBOW is used to quickly train word embeddings, and these embeddings are used to initialize the embeddings of some more complicated model. Usually, this is referred to as *pretraining embeddings*. It almost always helps performance a couple of percent.

The CBOW model is as follows. Given a target word w_i and an N context window on each side, $w_{i-1}, \ldots, w_{i-N}$ and

$w_{i+1}, \ldots, w_{i+N}$, referring to all context words collectively as C, CBOW tries to minimize

$$-\log p(w_i|C) = -\log \text{Softmax}(A(\sum_{w \in C} q_w) + b)$$

where q_w is the embedding for word w.

Implement this model in Pytorch by filling in the class below. Some tips:

- Think about which parameters you need to define.
- Make sure you know what shape each operation expects. Use .view() if you need to reshape.

```
CONTEXT_SIZE = 2  # 2 words to the left, 2 to the right
raw_text = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells.""".split()

# By deriving a set from `raw_text`, we deduplicate the array
vocab = set(raw_text)
vocab_size = len(vocab)

word_to_ix = {word: i for i, word in enumerate(vocab)}
data = []
for i in range(2, len(raw_text) - 2):
    context = [raw_text[i - 2], raw_text[i - 1],
               raw_text[i + 1], raw_text[i + 2]]
    target = raw_text[i]
    data.append((context, target))
print(data[:5])


class CBOW(nn.Module):

    def __init__(self):
        pass

    def forward(self, inputs):
        pass

# create your model and train.  here are some functions to help you make
# the data ready for use by your module


def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    tensor = torch.LongTensor(idxs)
    return autograd.Variable(tensor)


make_context_vector(data[0][0], word_to_ix)  # example
```

Out:

```
[(['We', 'are', 'to', 'study'], 'about'), (['are', 'about', 'study', 'the'], 'to'), ([
↪'about', 'to', 'the', 'idea'], 'study'), (['to', 'study', 'idea', 'of'], 'the'), ([
↪'study', 'the', 'of', 'a'], 'idea')]
```

Total running time of the script: (0 minutes 0.522 seconds)

Download Python source code: word_embeddings_tutorial.py

Download Jupyter notebook: word_embeddings_tutorial.ipynb

Generated by Sphinx-Gallery

1.6.4 Sequence Models and Long-Short Term Memory Networks

At this point, we have seen various feed-forward networks. That is, there is no state maintained by the network at all. This might not be the behavior we want. Sequence models are central to NLP: they are models where there is some sort of dependence through time between your inputs. The classical example of a sequence model is the Hidden Markov Model for part-of-speech tagging. Another example is the conditional random field.

A recurrent neural network is a network that maintains some kind of state. For example, its output could be used as part of the next input, so that information can propogate along as the network passes over the sequence. In the case of an LSTM, for each element in the sequence, there is a corresponding *hidden state* h_t, which in principle can contain information from arbitrary points earlier in the sequence. We can use the hidden state to predict words in a language model, part-of-speech tags, and a myriad of other things.

LSTM's in Pytorch

Before getting to the example, note a few things. Pytorch's LSTM expects all of its inputs to be 3D tensors. The semantics of the axes of these tensors is important. The first axis is the sequence itself, the second indexes instances in the mini-batch, and the third indexes elements of the input. We haven't discussed mini-batching, so lets just ignore that and assume we will always have just 1 dimension on the second axis. If we want to run the sequence model over the sentence "The cow jumped", our input should look like

$$\begin{bmatrix} \overbrace{q_{\text{The}}}^{\text{row vector}} \\ q_{\text{cow}} \\ q_{\text{jumped}} \end{bmatrix}$$

Except remember there is an additional 2nd dimension with size 1.

In addition, you could go through the sequence one at a time, in which case the 1st axis will have size 1 also.

Let's see a quick example.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

```
lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
inputs = [autograd.Variable(torch.randn((1, 3)))
          for _ in range(5)]  # make a sequence of length 5

# initialize the hidden state.
hidden = (autograd.Variable(torch.randn(1, 1, 3)),
          autograd.Variable(torch.randn((1, 1, 3))))
```

```
for i in inputs:
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden)

# alternatively, we can do the entire sequence all at once.
# the first value returned by LSTM is all of the hidden states throughout
# the sequence. the second is just the most recent hidden state
# (compare the last slice of "out" with "hidden" below, they are the same)
# The reason for this is that:
# "out" will give you access to all hidden states in the sequence
# "hidden" will allow you to continue the sequence and backpropagate,
# by passing it as an argument  to the lstm at a later time
# Add the extra 2nd dimension
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (autograd.Variable(torch.randn(1, 1, 3)), autograd.Variable(
    torch.randn((1, 1, 3))))  # clean out hidden state
out, hidden = lstm(inputs, hidden)
print(out)
print(hidden)
```

Out:

```
Variable containing:
(0 ,.,.) =
 -0.0187  0.1713 -0.2944

(1 ,.,.) =
 -0.3521  0.1026 -0.2971

(2 ,.,.) =
 -0.3191  0.0781 -0.1957

(3 ,.,.) =
 -0.1634  0.0941 -0.1637

(4 ,.,.) =
 -0.3368  0.0959 -0.0538
[torch.FloatTensor of size 5x1x3]

(Variable containing:
(0 ,.,.) =
 -0.3368  0.0959 -0.0538
[torch.FloatTensor of size 1x1x3]
, Variable containing:
(0 ,.,.) =
 -0.9825  0.4715 -0.0633
[torch.FloatTensor of size 1x1x3]
)
```

Example: An LSTM for Part-of-Speech Tagging

In this section, we will use an LSTM to get part of speech tags. We will not use Viterbi or Forward-Backward or anything like that, but as a (challenging) exercise to the reader, think about how Viterbi could be used after you have seen what is going on.

The model is as follows: let our input sentence be $w_1, \dots, w_M$, where $w_i \in V$, our vocab. Also, let T be our tag set,

and y_i the tag of word w_i. Denote our prediction of the tag of word w_i by $\hat{y}_i$.

This is a structure prediction, model, where our output is a sequence $\hat{y}_1, \ldots, \hat{y}_M$, where $\hat{y}_i \in T$.

To do the prediction, pass an LSTM over the sentence. Denote the hidden state at timestep i as h_i. Also, assign each tag a unique index (like how we had word_to_ix in the word embeddings section). Then our prediction rule for $\hat{y}_i$ is

$$\hat{y}_i = \text{argmax}_j \ (\log \text{Softmax}(Ah_i + b))_j$$

That is, take the log softmax of the affine map of the hidden state, and the predicted tag is the tag that has the maximum value in this vector. Note this implies immediately that the dimensionality of the target space of A is $|T|$.

Prepare data:

```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    tensor = torch.LongTensor(idxs)
    return autograd.Variable(tensor)


training_data = [
    ("The dog ate the apple".split(), ["DET", "NN", "V", "DET", "NN"]),
    ("Everybody read that book".split(), ["NN", "V", "DET", "NN"])
]
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
print(word_to_ix)
tag_to_ix = {"DET": 0, "NN": 1, "V": 2}

# These will usually be more like 32 or 64 dimensional.
# We will keep them small, so we can see how the weights change as we train.
EMBEDDING_DIM = 6
HIDDEN_DIM = 6
```

Out:

```
{'The': 0, 'dog': 1, 'ate': 2, 'the': 3, 'apple': 4, 'Everybody': 5, 'read': 6, 'that
↪': 7, 'book': 8}
```

Create the model:

```
class LSTMTagger(nn.Module):

    def __init__(self, embedding_dim, hidden_dim, vocab_size, tagset_size):
        super(LSTMTagger, self).__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs, and outputs hidden states
        # with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()
```

```
    def init_hidden(self):
        # Before we've done anything, we dont have any hidden state.
        # Refer to the Pytorch documentation to see exactly
        # why they have this dimensionality.
        # The axes semantics are (num_layers, minibatch_size, hidden_dim)
        return (autograd.Variable(torch.zeros(1, 1, self.hidden_dim)),
                autograd.Variable(torch.zeros(1, 1, self.hidden_dim)))

    def forward(self, sentence):
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(
            embeds.view(len(sentence), 1, -1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

Train the model:

```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM, len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# See what the scores are before training
# Note that element i,j of the output is the score for tag j for word i.
inputs = prepare_sequence(training_data[0][0], word_to_ix)
tag_scores = model(inputs)
print(tag_scores)

for epoch in range(300):  # again, normally you would NOT do 300 epochs, it is toy␣
↪data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Also, we need to clear out the hidden state of the LSTM,
        # detaching it from its history on the last instance.
        model.hidden = model.init_hidden()

        # Step 2. Get our inputs ready for the network, that is, turn them into
        # Variables of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = prepare_sequence(tags, tag_to_ix)

        # Step 3. Run our forward pass.
        tag_scores = model(sentence_in)

        # Step 4. Compute the loss, gradients, and update the parameters by
        #  calling optimizer.step()
        loss = loss_function(tag_scores, targets)
        loss.backward()
        optimizer.step()

# See what the scores are after training
inputs = prepare_sequence(training_data[0][0], word_to_ix)
tag_scores = model(inputs)
# The sentence is "the dog ate the apple".  i,j corresponds to score for tag j
```

```
#  for word i. The predicted tag is the maximum scoring tag.
# Here, we can see the predicted sequence below is 0 1 2 0 1
# since 0 is index of the maximum value of row 1,
# 1 is the index of maximum value of row 2, etc.
# Which is DET NOUN VERB DET NOUN, the correct sequence!
print(tag_scores)
```

Out:

```
Variable containing:
-1.1989 -0.9630 -1.1497
-1.2522 -0.9158 -1.1586
-1.2563 -1.0022 -1.0550
-1.1518 -1.1443 -1.0065
-1.1728 -1.0677 -1.0593
[torch.FloatTensor of size 5x3]

Variable containing:
-0.1902 -1.8654 -3.9957
-4.1051 -0.0263 -4.6590
-4.0204 -3.1797 -0.0614
-0.0372 -4.3504 -3.7448
-4.0387 -0.0348 -4.1001
[torch.FloatTensor of size 5x3]
```

Exercise: Augmenting the LSTM part-of-speech tagger with character-level features

In the example above, each word had an embedding, which served as the inputs to our sequence model. Let's augment the word embeddings with a representation derived from the characters of the word. We expect that this should help significantly, since character-level information like affixes have a large bearing on part-of-speech. For example, words with the affix *-ly* are almost always tagged as adverbs in English.

To do this, let c_w be the character-level representation of word w. Let x_w be the word embedding as before. Then the input to our sequence model is the concatenation of x_w and c_w. So if x_w has dimension 5, and c_w dimension 3, then our LSTM should accept an input of dimension 8.

To get the character level representation, do an LSTM over the characters of a word, and let c_w be the final hidden state of this LSTM. Hints:

- There are going to be two LSTM's in your new model. The original one that outputs POS tag scores, and the new one that outputs a character-level representation of each word.
- To do a sequence model over characters, you will have to embed characters. The character embeddings will be the input to the character LSTM.

Total running time of the script: (0 minutes 0.950 seconds)

Download Python source code: sequence_models_tutorial.py

Download Jupyter notebook: sequence_models_tutorial.ipynb

Generated by Sphinx-Gallery

1.6.5 Advanced: Making Dynamic Decisions and the Bi-LSTM CRF

Dynamic versus Static Deep Learning Toolkits

Pytorch is a *dynamic* neural network kit. Another example of a dynamic kit is Dynet (I mention this because working with Pytorch and Dynet is similar. If you see an example in Dynet, it will probably help you implement it in Pytorch). The opposite is the *static* tool kit, which includes Theano, Keras, TensorFlow, etc. The core difference is the following:

- In a static toolkit, you define a computation graph once, compile it, and then stream instances to it.
- In a dynamic toolkit, you define a computation graph *for each instance*. It is never compiled and is executed on-the-fly

Without a lot of experience, it is difficult to appreciate the difference. One example is to suppose we want to build a deep constituent parser. Suppose our model involves roughly the following steps:

- We build the tree bottom up
- Tag the root nodes (the words of the sentence)
- From there, use a neural network and the embeddings of the words to find combinations that form constituents. Whenever you form a new constituent, use some sort of technique to get an embedding of the constituent. In this case, our network architecture will depend completely on the input sentence. In the sentence "The green cat scratched the wall", at some point in the model, we will want to combine the span $(i, j, r) = (1, 3, \text{NP})$ (that is, an NP constituent spans word 1 to word 3, in this case "The green cat").

However, another sentence might be "Somewhere, the big fat cat scratched the wall". In this sentence, we will want to form the constituent $(2, 4, NP)$ at some point. The constituents we will want to form will depend on the instance. If we just compile the computation graph once, as in a static toolkit, it will be exceptionally difficult or impossible to program this logic. In a dynamic toolkit though, there isn't just 1 pre-defined computation graph. There can be a new computation graph for each instance, so this problem goes away.

Dynamic toolkits also have the advantage of being easier to debug and the code more closely resembling the host language (by that I mean that Pytorch and Dynet look more like actual Python code than Keras or Theano).

Bi-LSTM Conditional Random Field Discussion

For this section, we will see a full, complicated example of a Bi-LSTM Conditional Random Field for named-entity recognition. The LSTM tagger above is typically sufficient for part-of-speech tagging, but a sequence model like the CRF is really essential for strong performance on NER. Familiarity with CRF's is assumed. Although this name sounds scary, all the model is is a CRF but where an LSTM provides the features. This is an advanced model though, far more complicated than any earlier model in this tutorial. If you want to skip it, that is fine. To see if you're ready, see if you can:

- Write the recurrence for the viterbi variable at step i for tag k.
- Modify the above recurrence to compute the forward variables instead.
- Modify again the above recurrence to compute the forward variables in log-space (hint: log-sum-exp)

If you can do those three things, you should be able to understand the code below. Recall that the CRF computes a conditional probability. Let y be a tag sequence and x an input sequence of words. Then we compute

$$P(y|x) = \frac{\exp\left(\text{Score}(x, y)\right)}{\sum_{y'} \exp\left(\text{Score}(x, y')\right)}$$

Where the score is determined by defining some log potentials $\log \psi_i(x, y)$ such that

$$\text{Score}(x, y) = \sum_i \log \psi_i(x, y)$$

To make the partition function tractable, the potentials must look only at local features.

In the Bi-LSTM CRF, we define two kinds of potentials: emission and transition. The emission potential for the word at index i comes from the hidden state of the Bi-LSTM at timestep i. The transition scores are stored in a $|T|x|T|$ matrix $\mathbf{P}$, where T is the tag set. In my implementation, $\mathbf{P}_{j,k}$ is the score of transitioning to tag j from tag k. So:

$$\text{Score}(x, y) = \sum_i \log \psi_{\text{EMIT}}(y_i \to x_i) + \log \psi_{\text{TRANS}}(y_{i-1} \to y_i)$$

$$= \sum_i h_i[y_i] + \mathbf{P}_{y_i, y_{i-1}}$$

where in this second expression, we think of the tags as being assigned unique non-negative indices.

If the above discussion was too brief, you can check out this write up from Michael Collins on CRFs.

Implementation Notes

The example below implements the forward algorithm in log space to compute the partition function, and the viterbi algorithm to decode. Backpropagation will compute the gradients automatically for us. We don't have to do anything by hand.

The implementation is not optimized. If you understand what is going on, you'll probably quickly see that iterating over the next tag in the forward algorithm could probably be done in one big operation. I wanted to code to be more readable. If you want to make the relevant change, you could probably use this tagger for real tasks.

```
# Author: Robert Guthrie

import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim

torch.manual_seed(1)
```

Helper functions to make the code more readable.

```
def to_scalar(var):
    # returns a python float
    return var.view(-1).data.tolist()[0]


def argmax(vec):
    # return the argmax as a python int
    _, idx = torch.max(vec, 1)
    return to_scalar(idx)


def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    tensor = torch.LongTensor(idxs)
    return autograd.Variable(tensor)


# Compute log sum exp in a numerically stable way for the forward algorithm
def log_sum_exp(vec):
    max_score = vec[0, argmax(vec)]
    max_score_broadcast = max_score.view(1, -1).expand(1, vec.size()[1])
    return max_score + \
        torch.log(torch.sum(torch.exp(vec - max_score_broadcast)))
```

Create model

```
class BiLSTM_CRF(nn.Module):

    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.tagset_size = len(tag_to_ix)

        self.word_embeds = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # Maps the output of the LSTM into tag space.
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # Matrix of transition parameters.  Entry i,j is the score of
        # transitioning *to* i *from* j.
        self.transitions = nn.Parameter(
            torch.randn(self.tagset_size, self.tagset_size))

        # These two statements enforce the constraint that we never transfer
        # to the start tag and we never transfer from the stop tag
        self.transitions.data[tag_to_ix[START_TAG], :] = -10000
        self.transitions.data[:, tag_to_ix[STOP_TAG]] = -10000

        self.hidden = self.init_hidden()

    def init_hidden(self):
        return (autograd.Variable(torch.randn(2, 1, self.hidden_dim // 2)),
                autograd.Variable(torch.randn(2, 1, self.hidden_dim // 2)))

    def _forward_alg(self, feats):
        # Do the forward algorithm to compute the partition function
        init_alphas = torch.Tensor(1, self.tagset_size).fill_(-10000.)
        # START_TAG has all of the score.
        init_alphas[0][self.tag_to_ix[START_TAG]] = 0.

        # Wrap in a variable so that we will get automatic backprop
        forward_var = autograd.Variable(init_alphas)

        # Iterate through the sentence
        for feat in feats:
            alphas_t = []  # The forward variables at this timestep
            for next_tag in range(self.tagset_size):
                # broadcast the emission score: it is the same regardless of
                # the previous tag
                emit_score = feat[next_tag].view(
                    1, -1).expand(1, self.tagset_size)
                # the ith entry of trans_score is the score of transitioning to
                # next_tag from i
                trans_score = self.transitions[next_tag].view(1, -1)
                # The ith entry of next_tag_var is the value for the
                # edge (i -> next_tag) before we do log-sum-exp
                next_tag_var = forward_var + trans_score + emit_score
                # The forward variable for this tag is log-sum-exp of all the
```

```
                # scores.
                alphas_t.append(log_sum_exp(next_tag_var))
            forward_var = torch.cat(alphas_t).view(1, -1)
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        alpha = log_sum_exp(terminal_var)
        return alpha

    def _get_lstm_features(self, sentence):
        self.hidden = self.init_hidden()
        embeds = self.word_embeds(sentence).view(len(sentence), 1, -1)
        lstm_out, self.hidden = self.lstm(embeds, self.hidden)
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def _score_sentence(self, feats, tags):
        # Gives the score of a provided tag sequence
        score = autograd.Variable(torch.Tensor([0]))
        tags = torch.cat([torch.LongTensor([self.tag_to_ix[START_TAG]]), tags])
        for i, feat in enumerate(feats):
            score = score + \
                self.transitions[tags[i + 1], tags[i]] + feat[tags[i + 1]]
        score = score + self.transitions[self.tag_to_ix[STOP_TAG], tags[-1]]
        return score

    def _viterbi_decode(self, feats):
        backpointers = []

        # Initialize the viterbi variables in log space
        init_vvars = torch.Tensor(1, self.tagset_size).fill_(-10000.)
        init_vvars[0][self.tag_to_ix[START_TAG]] = 0

        # forward_var at step i holds the viterbi variables for step i-1
        forward_var = autograd.Variable(init_vvars)
        for feat in feats:
            bptrs_t = []  # holds the backpointers for this step
            viterbivars_t = []  # holds the viterbi variables for this step

            for next_tag in range(self.tagset_size):
                # next_tag_var[i] holds the viterbi variable for tag i at the
                # previous step, plus the score of transitioning
                # from tag i to next_tag.
                # We don't include the emission scores here because the max
                # does not depend on them (we add them in below)
                next_tag_var = forward_var + self.transitions[next_tag]
                best_tag_id = argmax(next_tag_var)
                bptrs_t.append(best_tag_id)
                viterbivars_t.append(next_tag_var[0][best_tag_id])
            # Now add in the emission scores, and assign forward_var to the set
            # of viterbi variables we just computed
            forward_var = (torch.cat(viterbivars_t) + feat).view(1, -1)
            backpointers.append(bptrs_t)

        # Transition to STOP_TAG
        terminal_var = forward_var + self.transitions[self.tag_to_ix[STOP_TAG]]
        best_tag_id = argmax(terminal_var)
        path_score = terminal_var[0][best_tag_id]
```

```
        # Follow the back pointers to decode the best path.
        best_path = [best_tag_id]
        for bptrs_t in reversed(backpointers):
            best_tag_id = bptrs_t[best_tag_id]
            best_path.append(best_tag_id)
        # Pop off the start tag (we dont want to return that to the caller)
        start = best_path.pop()
        assert start == self.tag_to_ix[START_TAG]  # Sanity check
        best_path.reverse()
        return path_score, best_path

    def neg_log_likelihood(self, sentence, tags):
        feats = self._get_lstm_features(sentence)
        forward_score = self._forward_alg(feats)
        gold_score = self._score_sentence(feats, tags)
        return forward_score - gold_score

    def forward(self, sentence):  # dont confuse this with _forward_alg above.
        # Get the emission scores from the BiLSTM
        lstm_feats = self._get_lstm_features(sentence)

        # Find the best path, given the features.
        score, tag_seq = self._viterbi_decode(lstm_feats)
        return score, tag_seq
```

Run training

```
START_TAG = "<START>"
STOP_TAG = "<STOP>"
EMBEDDING_DIM = 5
HIDDEN_DIM = 4

# Make up some training data
training_data = [(
    "the wall street journal reported today that apple corporation made money".
↪split(),
    "B I I I O O O B I O O".split()
), (
    "georgia tech is a university in georgia".split(),
    "B I O O O O B".split()
)]

word_to_ix = {}
for sentence, tags in training_data:
    for word in sentence:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)

tag_to_ix = {"B": 0, "I": 1, "O": 2, START_TAG: 3, STOP_TAG: 4}

model = BiLSTM_CRF(len(word_to_ix), tag_to_ix, EMBEDDING_DIM, HIDDEN_DIM)
optimizer = optim.SGD(model.parameters(), lr=0.01, weight_decay=1e-4)

# Check predictions before training
precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
precheck_tags = torch.LongTensor([tag_to_ix[t] for t in training_data[0][1]])
print(model(precheck_sent))
```

```
# Make sure prepare_sequence from earlier in the LSTM section is loaded
for epoch in range(
        300):  # again, normally you would NOT do 300 epochs, it is toy data
    for sentence, tags in training_data:
        # Step 1. Remember that Pytorch accumulates gradients.
        # We need to clear them out before each instance
        model.zero_grad()

        # Step 2. Get our inputs ready for the network, that is,
        # turn them into Variables of word indices.
        sentence_in = prepare_sequence(sentence, word_to_ix)
        targets = torch.LongTensor([tag_to_ix[t] for t in tags])

        # Step 3. Run our forward pass.
        neg_log_likelihood = model.neg_log_likelihood(sentence_in, targets)

        # Step 4. Compute the loss, gradients, and update the parameters by
        # calling optimizer.step()
        neg_log_likelihood.backward()
        optimizer.step()

# Check predictions after training
precheck_sent = prepare_sequence(training_data[0][0], word_to_ix)
print(model(precheck_sent))
# We got it!
```

Out:

```
(Variable containing:
 18.0485
[torch.FloatTensor of size 1]
, [0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0])
(Variable containing:
 20.7529
[torch.FloatTensor of size 1]
, [0, 1, 1, 1, 2, 2, 2, 0, 1, 2, 2])
```

Exercise: A new loss function for discriminative tagging

It wasn't really necessary for us to create a computation graph when doing decoding, since we do not backpropagate from the viterbi path score. Since we have it anyway, try training the tagger where the loss function is the difference between the Viterbi path score and the score of the gold-standard path. It should be clear that this function is non-negative and 0 when the predicted tag sequence is the correct tag sequence. This is essentially *structured perceptron*.

This modification should be short, since Viterbi and score_sentence are already implemented. This is an example of the shape of the computation graph *depending on the training instance*. Although I haven't tried implementing this in a static toolkit, I imagine that it is possible but much less straightforward.

Pick up some real data and do a comparison!

Total running time of the script: (0 minutes 12.384 seconds)

`Download Python source code: advanced_tutorial.py`

`Download Jupyter notebook: advanced_tutorial.ipynb`

Generated by Sphinx-Gallery

CHAPTER

TWO

INTERMEDIATE TUTORIALS

2.1 Classifying Names with a Character-Level RNN

Author: Sean Robertson

We will be building and training a basic character-level RNN to classify words. A character-level RNN reads words as a series of characters - outputting a prediction and "hidden state" at each step, feeding its previous hidden state into each next step. We take the final prediction to be the output, i.e. which class the word belongs to.

Specifically, we'll train on a few thousand surnames from 18 languages of origin, and predict which language a name is from based on the spelling:

```
$ python predict.py Hinton
(-0.47) Scottish
(-1.52) English
(-3.57) Irish

$ python predict.py Schmidhuber
(-0.19) German
(-2.48) Czech
(-2.68) Dutch
```

Recommended Reading:

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- http://pytorch.org/ For installation instructions
- *Deep Learning with PyTorch: A 60 Minute Blitz* to get started with PyTorch in general
- *Learning PyTorch with Examples* for a wide and deep overview
- *PyTorch for former Torch users* if you are former Lua Torch user

It would also be useful to know about RNNs and how they work:

- The Unreasonable Effectiveness of Recurrent Neural Networks shows a bunch of real life examples
- Understanding LSTM Networks is about LSTMs specifically but also informative about RNNs in general

2.1.1 Preparing the Data

Note: Download the data from here and extract it to the current directory.

Included in the `data/names` directory are 18 text files named as "[Language].txt". Each file contains a bunch of names, one name per line, mostly romanized (but we still need to convert from Unicode to ASCII).

We'll end up with a dictionary of lists of names per language, `{language: [names ...]}`. The generic variables "category" and "line" (for language and name in our case) are used for later extensibility.

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob

def findFiles(path): return glob.glob(path)

print(findFiles('data/names/*.txt'))

import unicodedata
import string

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

# Turn a Unicode string to plain ASCII, thanks to http://stackoverflow.com/a/518232/
↪2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print(unicodeToAscii('Ślusàrski'))

# Build the category_lines dictionary, a list of names per language
category_lines = {}
all_categories = []

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

for filename in findFiles('data/names/*.txt'):
    category = filename.split('/')[-1].split('.')[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

Out:

```
['data/names/Czech.txt', 'data/names/German.txt', 'data/names/Arabic.txt', 'data/
↪names/Japanese.txt', 'data/names/Chinese.txt', 'data/names/Vietnamese.txt', 'data/
↪names/Russian.txt', 'data/names/French.txt', 'data/names/Irish.txt', 'data/names/
↪English.txt', 'data/names/Spanish.txt', 'data/names/Greek.txt', 'data/names/Italian.
↪txt', 'data/names/Portuguese.txt', 'data/names/Scottish.txt', 'data/names/Dutch.txt
↪', 'data/names/Korean.txt', 'data/names/Polish.txt']
Slusarski
```

Now we have `category_lines`, a dictionary mapping each category (language) to a list of lines (names). We also

kept track of `all_categories` (just a list of languages) and `n_categories` for later reference.

```
print(category_lines['Italian'][:5])
```

Out:

```
['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
```

Turning Names into Tensors

Now that we have all the names organized, we need to turn them into Tensors to make any use of them.

To represent a single letter, we use a "one-hot vector" of size `<1 x n_letters>`. A one-hot vector is filled with 0s except for a 1 at index of the current letter, e.g. `"b" = <0 1 0 0 0 ...>`.

To make a word we join a bunch of those into a 2D matrix `<line_length x 1 x n_letters>`.

That extra 1 dimension is because PyTorch assumes everything is in batches - we're just using a batch size of 1 here.

```
import torch

# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# Just for demonstration, turn a letter into a <1 x n_letters> Tensor
def letterToTensor(letter):
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1
    return tensor

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

print(letterToTensor('J'))

print(lineToTensor('Jones').size())
```

Out:

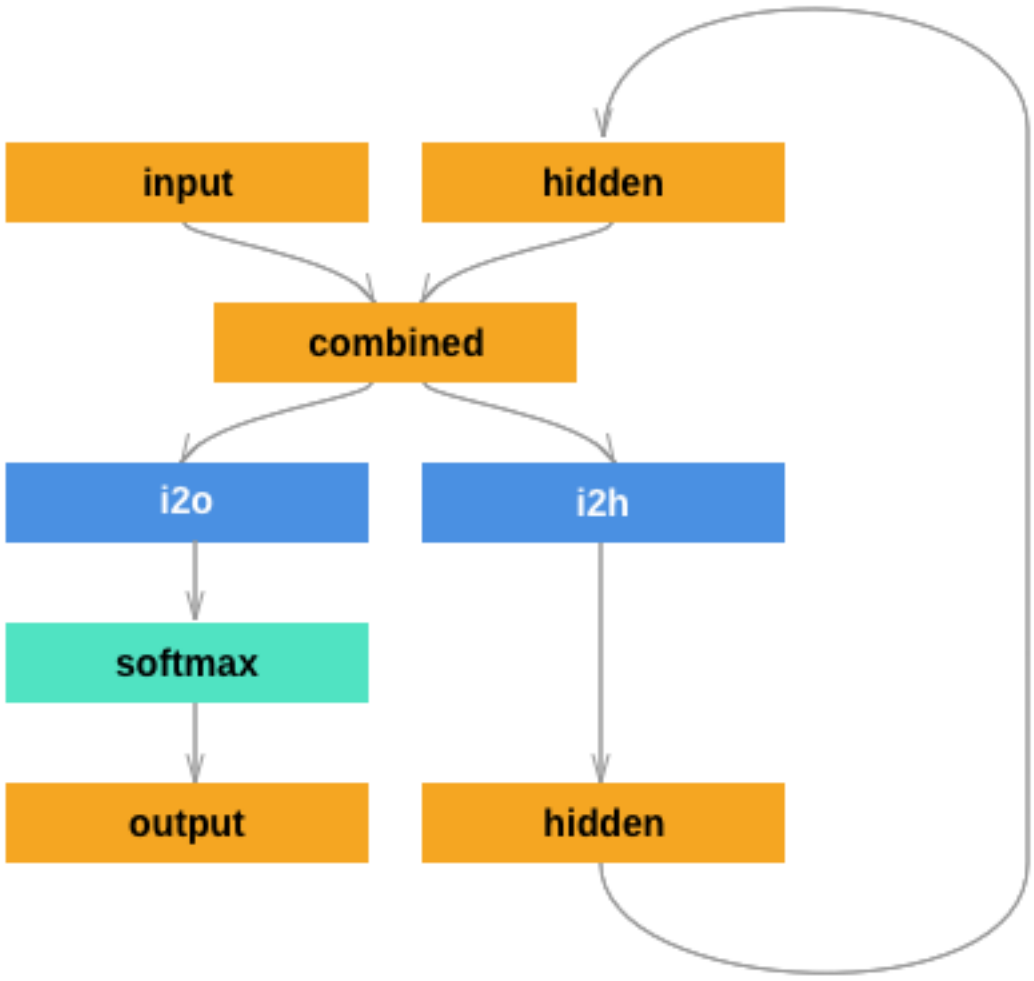
```
Columns 0 to 12
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 13 to 25
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 26 to 38
    0     0     0     0     0     0     0     0     0     1     0     0     0

Columns 39 to 51
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 52 to 56
```

```
    0     0     0     0     0
[torch.FloatTensor of size 1x57]

torch.Size([5, 1, 57])
```

2.1.2 Creating the Network

Before autograd, creating a recurrent neural network in Torch involved cloning the parameters of a layer over several timesteps. The layers held hidden state and gradients which are now entirely handled by the graph itself. This means you can implement a RNN in a very "pure" way, as regular feed-forward layers.

This RNN module (mostly copied from the PyTorch for Torch users tutorial) is just 2 linear layers which operate on an input and hidden state, with a LogSoftmax layer after the output.




```
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)
        self.softmax = nn.LogSoftmax(dim=1)

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

    def initHidden(self):
        return Variable(torch.zeros(1, self.hidden_size))

n_hidden = 128
rnn = RNN(n_letters, n_hidden, n_categories)
```

To run a step of this network we need to pass an input (in our case, the Tensor for the current letter) and a previous hidden state (which we initialize as zeros at first). We'll get back the output (probability of each language) and a next hidden state (which we keep for the next step).

Remember that PyTorch modules operate on Variables rather than straight up Tensors.

```
input = Variable(letterToTensor('A'))
hidden = Variable(torch.zeros(1, n_hidden))

output, next_hidden = rnn(input, hidden)
```

For the sake of efficiency we don't want to be creating a new Tensor for every step, so we will use `lineToTensor` instead of `letterToTensor` and use slices. This could be further optimized by pre-computing batches of Tensors.

```
input = Variable(lineToTensor('Albert'))
hidden = Variable(torch.zeros(1, n_hidden))

output, next_hidden = rnn(input[0], hidden)
print(output)
```

Out:

```
Variable containing:

Columns 0 to 9
-2.8039 -2.9628 -2.8634 -2.9441 -2.8836 -2.8385 -2.9236 -2.8231 -2.9270 -2.8344

Columns 10 to 17
-2.9410 -2.9589 -2.8472 -2.8181 -2.9682 -2.9291 -2.9299 -2.8568
[torch.FloatTensor of size 1x18]
```

As you can see the output is a `<1 x n_categories>` Tensor, where every item is the likelihood of that category (higher is more likely).

2.1.3 Training

Preparing for Training

Before going into training we should make a few helper functions. The first is to interpret the output of the network, which we know to be a likelihood of each category. We can use `Tensor.topk` to get the index of the greatest value:

```
def categoryFromOutput(output):
    top_n, top_i = output.data.topk(1) # Tensor out of Variable with .data
    category_i = top_i[0][0]
    return all_categories[category_i], category_i

print(categoryFromOutput(output))
```

Out:

```
('Czech', 0)
```

We will also want a quick way to get a training example (a name and its language):

```
import random

def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    category_tensor = Variable(torch.LongTensor([all_categories.index(category)]))
    line_tensor = Variable(lineToTensor(line))
    return category, line, category_tensor, line_tensor

for i in range(10):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    print('category =', category, '/ line =', line)
```

Out:

```
category = Russian / line = Balanowsky
category = Spanish / line = Tosell
category = Russian / line = Omelianovsky
category = Chinese / line = Loong
category = Portuguese / line = Delgado
category = Czech / line = Michalovicova
category = Greek / line = Adamou
category = Italian / line = Aliberti
category = Irish / line = Damhan
category = Spanish / line = Salazar
```

Training the Network

Now all it takes to train this network is show it a bunch of examples, have it make guesses, and tell it if it's wrong.

For the loss function `nn.NLLLoss` is appropriate, since the last layer of the RNN is `nn.LogSoftmax`.

```
criterion = nn.NLLLoss()
```

Each loop of training will:

- Create input and target tensors

- Create a zeroed initial hidden state
- Read each letter in and
  - Keep hidden state for next letter
- Compare final output to target
- Back-propagate
- Return the output and loss

```
learning_rate = 0.005 # If you set this too high, it might explode. If too low, it
↪might not learn

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

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

Now we just have to run that with a bunch of examples. Since the `train` function returns both the output and loss we can print its guesses and also keep track of loss for plotting. Since there are 1000s of examples we print only every `print_every` examples, and take an average of the loss.

```
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000



# Keep track of losses for plotting
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    category, line, category_tensor, line_tensor = randomTrainingExample()
```

```
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else ' (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100,␣
↪timeSince(start), loss, line, guess, correct))

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

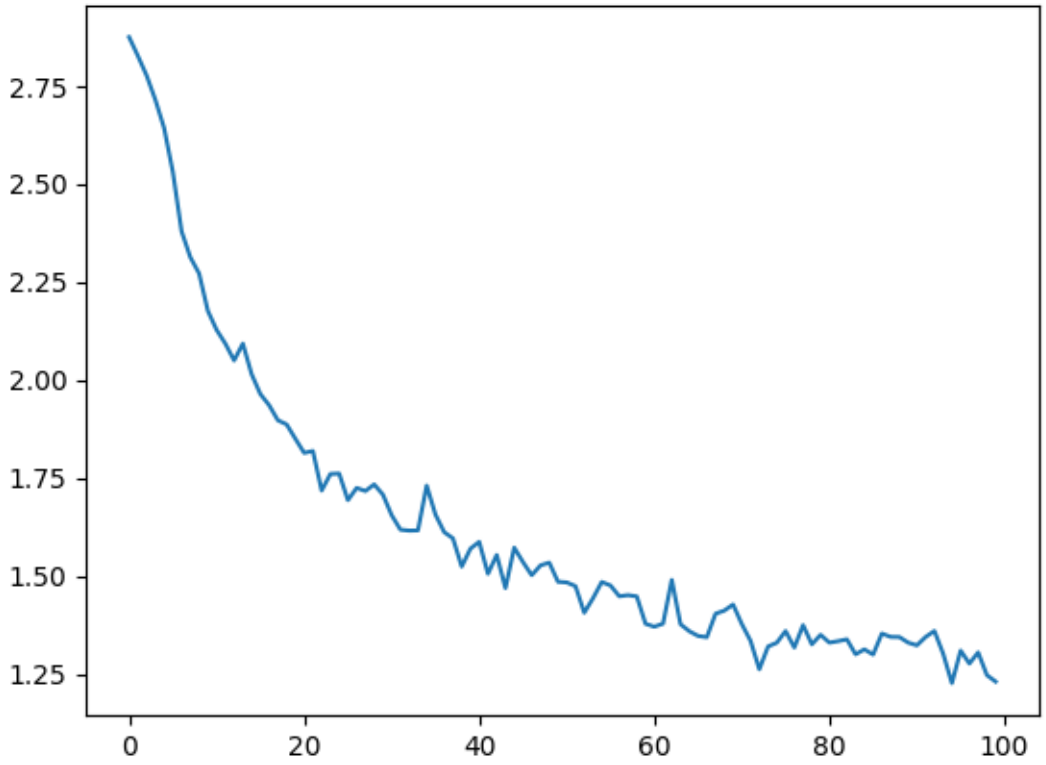
Out:

```
5000 5% (0m 6s) 2.9799 Rompay / Russian  (Dutch)
10000 10% (0m 12s) 3.4983 Rompaij / Irish  (Dutch)
15000 15% (0m 18s) 2.4390 Williamson / Russian  (Scottish)
20000 20% (0m 25s) 2.5871 Costa / Czech  (Portuguese)
25000 25% (0m 31s) 0.7670 Baldi / Italian ✓
30000 30% (0m 37s) 2.2803 Delgado / Italian  (Portuguese)
35000 35% (0m 43s) 0.4095 Belloni / Italian ✓
40000 40% (0m 49s) 1.6705 Hamraev / Russian ✓
45000 45% (0m 55s) 0.2133 Yoon / Korean ✓
50000 50% (1m 2s) 0.6972 Aswad / Arabic ✓
55000 55% (1m 8s) 0.7448 Huan / Chinese ✓
60000 60% (1m 14s) 0.2492 Khouri / Arabic ✓
65000 65% (1m 20s) 1.9285 Nifterick / German  (Dutch)
70000 70% (1m 26s) 1.8211 Castellano / Irish  (Spanish)
75000 75% (1m 32s) 0.5865 Schuster / German ✓
80000 80% (1m 39s) 0.6497 Zavala / Spanish ✓
85000 85% (1m 45s) 0.3201 Kouros / Greek ✓
90000 90% (1m 51s) 0.0770 Yang / Korean ✓
95000 95% (1m 57s) 0.2934 Murray / Scottish ✓
100000 100% (2m 3s) 1.0676 Tahan / Arabic ✓
```

Plotting the Results

Plotting the historical loss from `all_losses` shows the network learning:

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
```

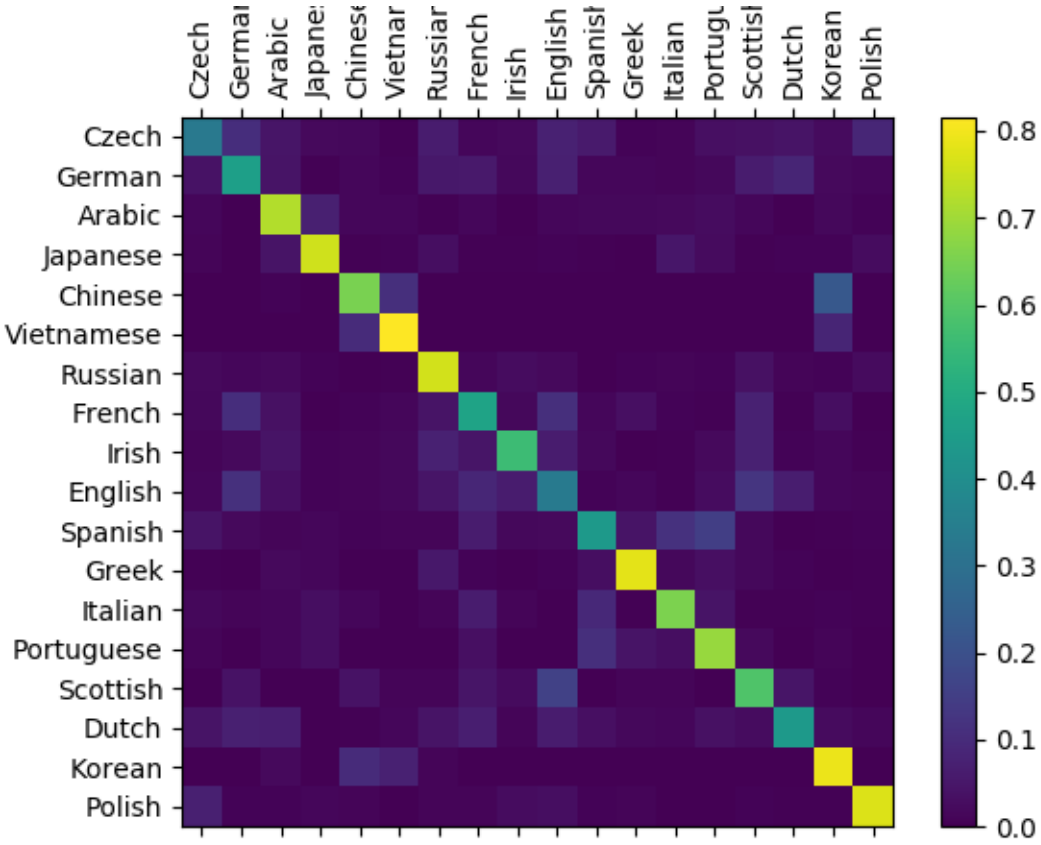


2.1.4 Evaluating the Results

To see how well the network performs on different categories, we will create a confusion matrix, indicating for every actual language (rows) which language the network guesses (columns). To calculate the confusion matrix a bunch of samples are run through the network with `evaluate()`, which is the same as `train()` minus the backprop.

```
# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# Just return an output given a line
def evaluate(line_tensor):
    hidden = rnn.initHidden()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output = evaluate(line_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = all_categories.index(category)
```

```
    confusion[category_i][guess_i] += 1

# Normalize by dividing every row by its sum
for i in range(n_categories):
    confusion[i] = confusion[i] / confusion[i].sum()

# Set up plot
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)

# Set up axes
ax.set_xticklabels([''] + all_categories, rotation=90)
ax.set_yticklabels([''] + all_categories)

# Force label at every tick
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

# sphinx_gallery_thumbnail_number = 2
plt.show()
```



You can pick out bright spots off the main axis that show which languages it guesses incorrectly, e.g. Chinese for Korean, and Spanish for Italian. It seems to do very well with Greek, and very poorly with English (perhaps because of overlap with other languages).

Running on User Input

```
def predict(input_line, n_predictions=3):
    print('\n> %s' % input_line)
    output = evaluate(Variable(lineToTensor(input_line)))

    # Get top N categories
    topv, topi = output.data.topk(n_predictions, 1, True)
    predictions = []

    for i in range(n_predictions):
        value = topv[0][i]
        category_index = topi[0][i]
        print('(%.2f) %s' % (value, all_categories[category_index]))
        predictions.append([value, all_categories[category_index]])

predict('Dovesky')
predict('Jackson')
predict('Satoshi')
```

Out:

```
> Dovesky
(-0.24) Russian
(-1.85) Czech
(-3.35) English

> Jackson
(-0.48) Scottish
(-1.52) English
(-2.76) Russian

> Satoshi
(-0.73) Italian
(-1.76) Japanese
(-2.74) Arabic
```

The final versions of the scripts in the Practical PyTorch repo split the above code into a few files:

- `data.py` (loads files)
- `model.py` (defines the RNN)
- `train.py` (runs training)
- `predict.py` (runs `predict()` with command line arguments)
- `server.py` (serve prediction as a JSON API with bottle.py)

Run `train.py` to train and save the network.

Run `predict.py` with a name to view predictions:

```
$ python predict.py Hazaki
(-0.42) Japanese
(-1.39) Polish
(-3.51) Czech
```

Run `server.py` and visit http://localhost:5533/Yourname to get JSON output of predictions.

2.1.5 Exercises

- Try with a different dataset of line -> category, for example:
  - Any word -> language
  - First name -> gender
  - Character name -> writer
  - Page title -> blog or subreddit
- Get better results with a bigger and/or better shaped network
  - Add more linear layers
  - Try the `nn.LSTM` and `nn.GRU` layers
  - Combine multiple of these RNNs as a higher level network

Total running time of the script: (2 minutes 9.749 seconds)

`Download Python source code: char_rnn_classification_tutorial.py`

`Download Jupyter notebook: char_rnn_classification_tutorial.ipynb`

Generated by Sphinx-Gallery

2.2 Generating Names with a Character-Level RNN

Author: Sean Robertson

In the *last tutorial* we used a RNN to classify names into their language of origin. This time we'll turn around and generate names from languages.

```
> python sample.py Russian RUS
Rovakov
Uantov
Shavakov

> python sample.py German GER
Gerren
Ereng
Rosher

> python sample.py Spanish SPA
Salla
Parer
Allan

> python sample.py Chinese CHI
Chan
Hang
Iun
```

We are still hand-crafting a small RNN with a few linear layers. The big difference is instead of predicting a category after reading in all the letters of a name, we input a category and output one letter at a time. Recurrently predicting characters to form language (this could also be done with words or other higher order constructs) is often referred to as a "language model".

Recommended Reading:

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- http://pytorch.org/ For installation instructions
- *Deep Learning with PyTorch: A 60 Minute Blitz* to get started with PyTorch in general
- *Learning PyTorch with Examples* for a wide and deep overview
- *PyTorch for former Torch users* if you are former Lua Torch user

It would also be useful to know about RNNs and how they work:

- The Unreasonable Effectiveness of Recurrent Neural Networks shows a bunch of real life examples
- Understanding LSTM Networks is about LSTMs specifically but also informative about RNNs in general

I also suggest the previous tutorial, *Classifying Names with a Character-Level RNN*

2.2.1 Preparing the Data

Note: Download the data from here and extract it to the current directory.

See the last tutorial for more detail of this process. In short, there are a bunch of plain text files `data/names/[Language].txt` with a name per line. We split lines into an array, convert Unicode to ASCII, and end up with a dictionary `{language: [names ...]}`.

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import unicodedata
import string

all_letters = string.ascii_letters + " .,;'-"
n_letters = len(all_letters) + 1 # Plus EOS marker

def findFiles(path): return glob.glob(path)

# Turn a Unicode string to plain ASCII, thanks to http://stackoverflow.com/a/518232/
↪2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

# Build the category_lines dictionary, a list of lines per category
category_lines = {}
all_categories = []
for filename in findFiles('data/names/*.txt'):
    category = filename.split('/')[-1].split('.')[0]
    all_categories.append(category)
    lines = readLines(filename)
```

```
    category_lines[category] = lines

n_categories = len(all_categories)

print('# categories:', n_categories, all_categories)
print(unicodeToAscii("O'Néàl"))
```

Out:

```
# categories: 18 ['Czech', 'German', 'Arabic', 'Japanese', 'Chinese', 'Vietnamese',
↪'Russian', 'French', 'Irish', 'English', 'Spanish', 'Greek', 'Italian', 'Portuguese
↪', 'Scottish', 'Dutch', 'Korean', 'Polish']
O'Neal
```

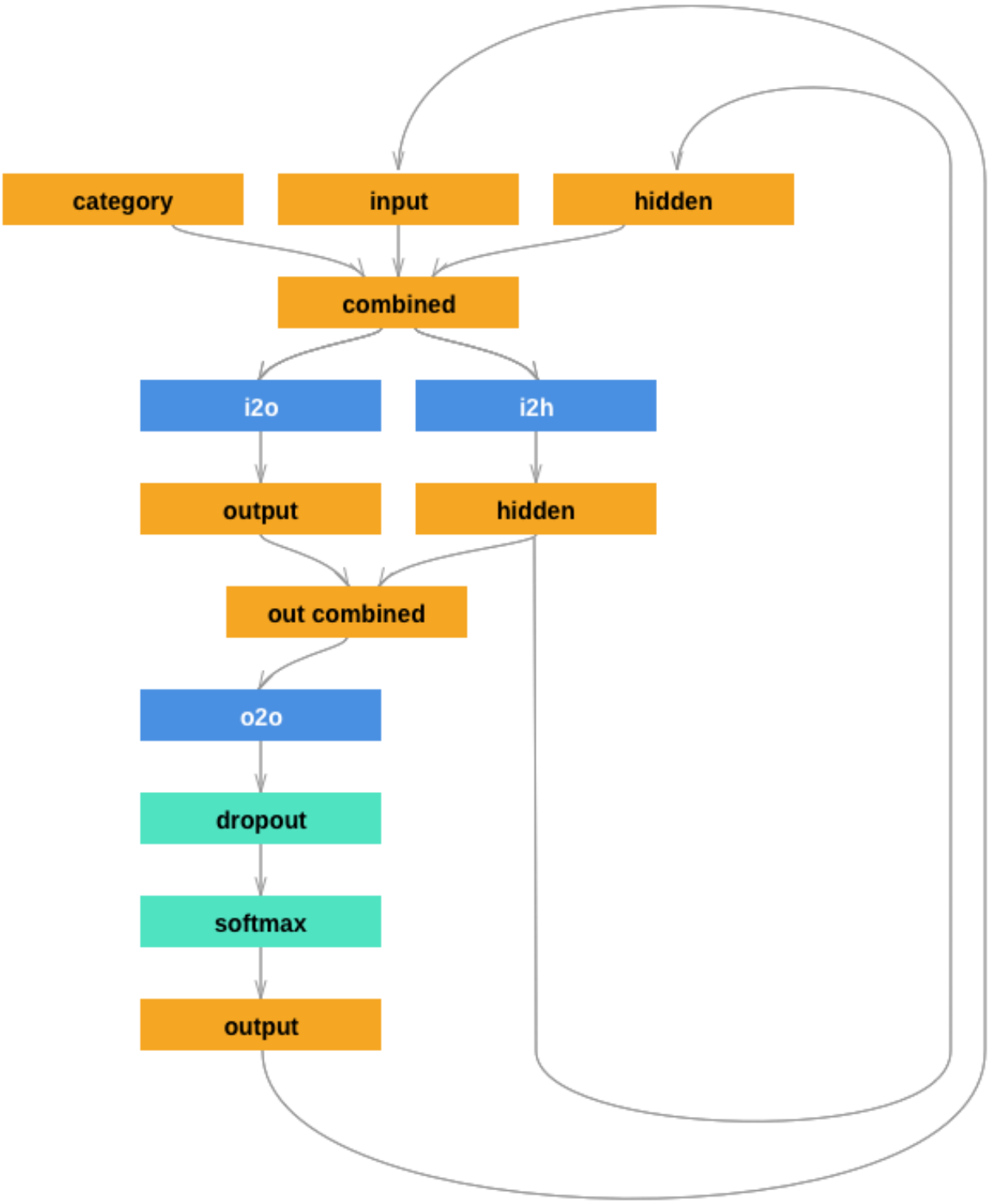
2.2.2 Creating the Network

This network extends *the last tutorial's RNN* with an extra argument for the category tensor, which is concatenated along with the others. The category tensor is a one-hot vector just like the letter input.

We will interpret the output as the probability of the next letter. When sampling, the most likely output letter is used as the next input letter.

I added a second linear layer `o2o` (after combining hidden and output) to give it more muscle to work with. There's also a dropout layer, which randomly zeros parts of its input with a given probability (here 0.1) and is usually used to fuzz inputs to prevent overfitting. Here we're using it towards the end of the network to purposely add some chaos and increase sampling variety.

```
import torch
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(n_categories + input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(n_categories + input_size + hidden_size, output_size)
        self.o2o = nn.Linear(hidden_size + output_size, output_size)
        self.dropout = nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, category, input, hidden):
        input_combined = torch.cat((category, input, hidden), 1)
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return Variable(torch.zeros(1, self.hidden_size))
```


category
input
hidden
combined
i2o
i2h
output
hidden
out combined
o2o
dropout
softmax
output

2.2.3 Training

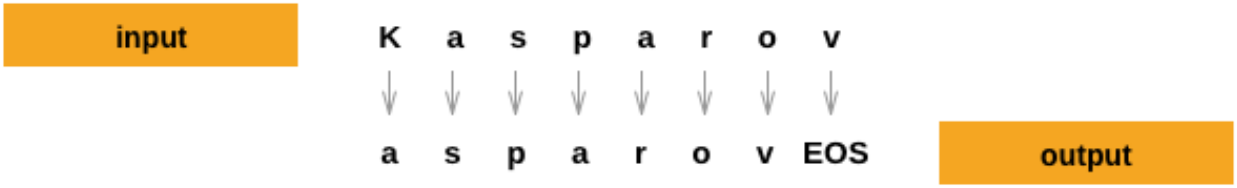
Preparing for Training

First of all, helper functions to get random pairs of (category, line):

```
import random

# Random item from a list
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

# Get a random category and random line from that category
def randomTrainingPair():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    return category, line
```

For each timestep (that is, for each letter in a training word) the inputs of the network will be `(category, current letter, hidden state)` and the outputs will be `(next letter, next hidden state)`. So for each training set, we'll need the category, a set of input letters, and a set of output/target letters.

Since we are predicting the next letter from the current letter for each timestep, the letter pairs are groups of consecutive letters from the line - e.g. for `"ABCD<EOS>"` we would create ("A", "B"), ("B", "C"), ("C", "D"), ("D", "EOS").



The category tensor is a one-hot tensor of size `<1 x n_categories>`. When training we feed it to the network at every timestep - this is a design choice, it could have been included as part of initial hidden state or some other strategy.

```
# One-hot vector for category
def categoryTensor(category):
    li = all_categories.index(category)
    tensor = torch.zeros(1, n_categories)
    tensor[0][li] = 1
    return tensor

# One-hot matrix of first to last letters (not including EOS) for input
def inputTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li in range(len(line)):
        letter = line[li]
        tensor[li][0][all_letters.find(letter)] = 1
    return tensor

# LongTensor of second letter to end (EOS) for target
def targetTensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
    letter_indexes.append(n_letters - 1) # EOS
    return torch.LongTensor(letter_indexes)
```

For convenience during training we'll make a `randomTrainingExample` function that fetches a random (category, line) pair and turns them into the required (category, input, target) tensors.

```
# Make category, input, and target tensors from a random category, line pair
def randomTrainingExample():
    category, line = randomTrainingPair()
    category_tensor = Variable(categoryTensor(category))
    input_line_tensor = Variable(inputTensor(line))
    target_line_tensor = Variable(targetTensor(line))
    return category_tensor, input_line_tensor, target_line_tensor
```

Training the Network

In contrast to classification, where only the last output is used, we are making a prediction at every step, so we are calculating loss at every step.

The magic of autograd allows you to simply sum these losses at each step and call backward at the end.

```
criterion = nn.NLLLoss()

learning_rate = 0.0005

def train(category_tensor, input_line_tensor, target_line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    loss = 0

    for i in range(input_line_tensor.size()[0]):
        output, hidden = rnn(category_tensor, input_line_tensor[i], hidden)
        loss += criterion(output, target_line_tensor[i])

    loss.backward()

    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.data[0] / input_line_tensor.size()[0]
```

To keep track of how long training takes I am adding a `timeSince(timestamp)` function which returns a human readable string:

```
import time
import math

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

Training is business as usual - call train a bunch of times and wait a few minutes, printing the current time and loss every `print_every` examples, and keeping store of an average loss per `plot_every` examples in `all_losses` for plotting later.

```
rnn = RNN(n_letters, 128, n_letters)

n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # Reset every plot_every iters

start = time.time()

for iter in range(1, n_iters + 1):
    output, loss = train(*randomTrainingExample())
    total_loss += loss

    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f' % (timeSince(start), iter, iter / n_iters * 100,␣
↪loss))

    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0
```

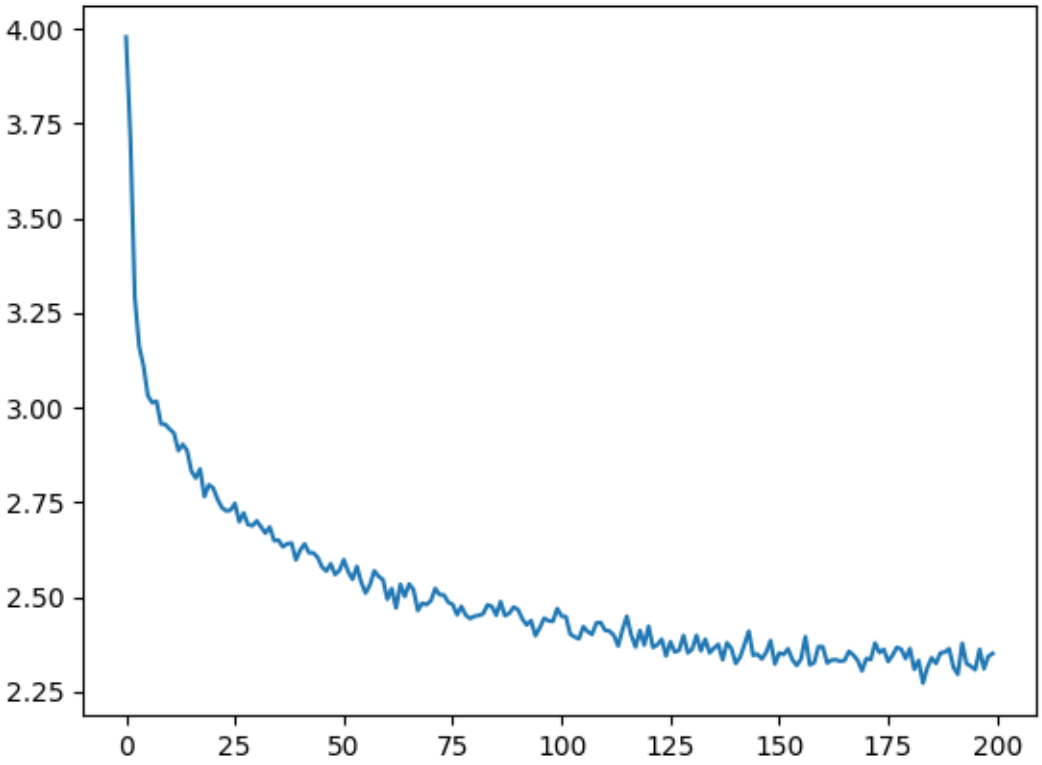
Out:

```
0m 13s (5000 5%) 2.3916
0m 27s (10000 10%) 2.9375
0m 41s (15000 15%) 2.5119
0m 54s (20000 20%) 2.1419
1m 8s (25000 25%) 2.7855
1m 22s (30000 30%) 2.3315
1m 37s (35000 35%) 2.1455
1m 51s (40000 40%) 2.2973
2m 9s (45000 45%) 2.6948
2m 24s (50000 50%) 2.2530
2m 38s (55000 55%) 2.9123
2m 53s (60000 60%) 1.6011
3m 7s (65000 65%) 1.7502
3m 21s (70000 70%) 2.2053
3m 35s (75000 75%) 2.3067
3m 50s (80000 80%) 2.6769
4m 5s (85000 85%) 2.4789
4m 21s (90000 90%) 2.5689
12m 7s (95000 95%) 2.8414
12m 26s (100000 100%) 3.0887
```

Plotting the Losses

Plotting the historical loss from all_losses shows the network learning:

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
```



2.2.4 Sampling the Network

To sample we give the network a letter and ask what the next one is, feed that in as the next letter, and repeat until the EOS token.

- Create tensors for input category, starting letter, and empty hidden state
- Create a string `output_name` with the starting letter
- Up to a maximum output length,
  - Feed the current letter to the network
  - Get the next letter from highest output, and next hidden state
  - If the letter is EOS, stop here
  - If a regular letter, add to `output_name` and continue
- Return the final name

Note: Rather than having to give it a starting letter, another strategy would have been to include a "start of string" token in training and have the network choose its own starting letter.

```
max_length = 20
```

```
# Sample from a category and starting letter
def sample(category, start_letter='A'):
    category_tensor = Variable(categoryTensor(category))
    input = Variable(inputTensor(start_letter))
    hidden = rnn.initHidden()

    output_name = start_letter

    for i in range(max_length):
        output, hidden = rnn(category_tensor, input[0], hidden)
        topv, topi = output.data.topk(1)
        topi = topi[0][0]
        if topi == n_letters - 1:
            break
        else:
            letter = all_letters[topi]
            output_name += letter
        input = Variable(inputTensor(letter))

    return output_name

# Get multiple samples from one category and multiple starting letters
def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
        print(sample(category, start_letter))

samples('Russian', 'RUS')

samples('German', 'GER')

samples('Spanish', 'SPA')

samples('Chinese', 'CHI')
```

Out:

```
Rovake
Uaran
Sanaka
Geren
Eren
Roure
Sana
Para
Alana
Cha
Han
Iun
```

2.2.5 Exercises

- Try with a different dataset of category -> line, for example:
  - Fictional series -> Character name
  - Part of speech -> Word
  - Country -> City

- Use a "start of sentence" token so that sampling can be done without choosing a start letter
- Get better results with a bigger and/or better shaped network
  - Try the nn.LSTM and nn.GRU layers
  - Combine multiple of these RNNs as a higher level network

Total running time of the script: (12 minutes 26.272 seconds)

`Download Python source code: char_rnn_generation_tutorial.py`

`Download Jupyter notebook: char_rnn_generation_tutorial.ipynb`

Generated by Sphinx-Gallery

2.3 Translation with a Sequence to Sequence Network and Attention

Author: Sean Robertson

In this project we will be teaching a neural network to translate from French to English.

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?

> elle n est pas poete mais romanciere .
= she is not a poet but a novelist .
< she not not a poet but a novelist .

> vous etes trop maigre .
= you re too skinny .
< you re all alone .
```

... to varying degrees of success.

This is made possible by the simple but powerful idea of the sequence to sequence network, in which two recurrent neural networks work together to transform one sequence to another. An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence.

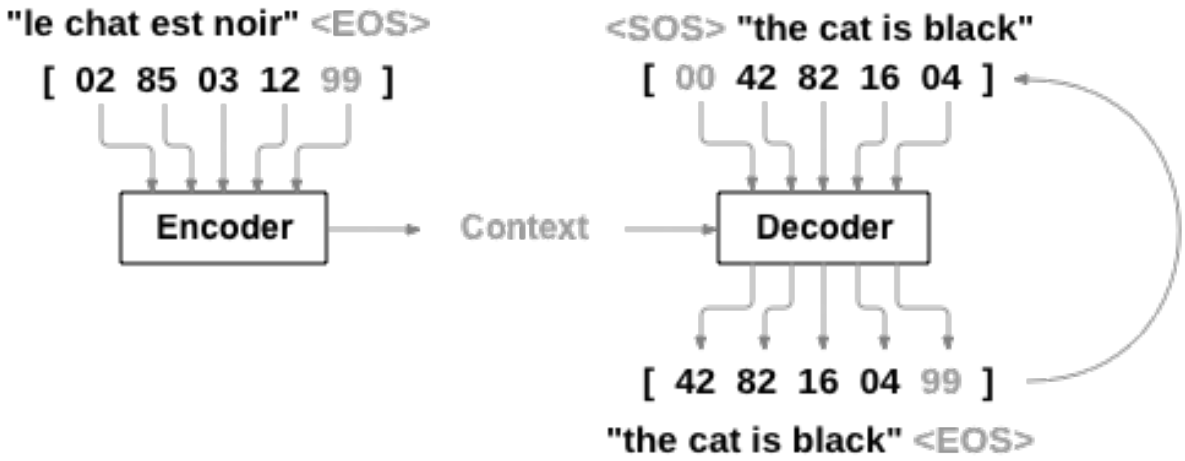
To improve upon this model we'll use an attention mechanism, which lets the decoder learn to focus over a specific range of the input sequence.

Recommended Reading:

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- http://pytorch.org/ For installation instructions
- *Deep Learning with PyTorch: A 60 Minute Blitz* to get started with PyTorch in general
- *Learning PyTorch with Examples* for a wide and deep overview
- *PyTorch for former Torch users* if you are former Lua Torch user

It would also be useful to know about Sequence to Sequence networks and how they work:



- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation by Jointly Learning to Align and Translate
- A Neural Conversational Model

You will also find the previous tutorials on *Classifying Names with a Character-Level RNN* and *Generating Names with a Character-Level RNN* helpful as those concepts are very similar to the Encoder and Decoder models, respectively.

And for more, read the papers that introduced these topics:

- Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation by Jointly Learning to Align and Translate
- A Neural Conversational Model

Requirements

```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch import optim
import torch.nn.functional as F

use_cuda = torch.cuda.is_available()
```

2.3.1 Loading data files

The data for this project is a set of many thousands of English to French translation pairs.

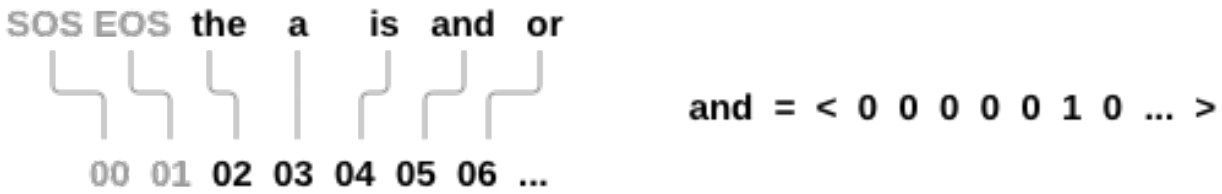
This question on Open Data Stack Exchange pointed me to the open translation site http://tatoeba.org/ which has downloads available at http://tatoeba.org/eng/downloads - and better yet, someone did the extra work of splitting language pairs into individual text files here: http://www.manythings.org/anki/

The English to French pairs are too big to include in the repo, so download to `data/eng-fra.txt` before continuing. The file is a tab separated list of translation pairs:

```
I am cold.    Je suis froid.
```

Note: Download the data from here and extract it to the current directory.

Similar to the character encoding used in the character-level RNN tutorials, we will be representing each word in a language as a one-hot vector, or giant vector of zeros except for a single one (at the index of the word). Compared to the dozens of characters that might exist in a language, there are many many more words, so the encoding vector is much larger. We will however cheat a bit and trim the data to only use a few thousand words per language.



We'll need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we will use a helper class called `Lang` which has word → index (`word2index`) and index → word (`index2word`) dictionaries, as well as a count of each word `word2count` to use to later replace rare words.

```
SOS_token = 0
EOS_token = 1


class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2  # Count SOS and EOS

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

The files are all in Unicode, to simplify we will turn Unicode characters to ASCII, make everything lowercase, and trim most punctuation.

```
# Turn a Unicode string to plain ASCII, thanks to
# http://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

To read the data file we will split the file into lines, and then split lines into pairs. The files are all English → Other Language, so if we want to translate from Other Language → English I added the `reverse` flag to reverse the pairs.

```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]

    # Reverse pairs, make Lang instances
    if reverse:
        pairs = [list(reversed(p)) for p in pairs]
        input_lang = Lang(lang2)
        output_lang = Lang(lang1)
    else:
        input_lang = Lang(lang1)
        output_lang = Lang(lang2)

    return input_lang, output_lang, pairs
```

Since there are a *lot* of example sentences and we want to train something quickly, we'll trim the data set to only relatively short and simple sentences. Here the maximum length is 10 words (that includes ending punctuation) and we're filtering to sentences that translate to the form "I am" or "He is" etc. (accounting for apostrophes replaced earlier).

```
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
```

```
)


def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

The full process for preparing the data is:

- Read text file and split into lines, split lines into pairs
- Normalize text, filter by length and content
- Make word lists from sentences in pairs

```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs


input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
```

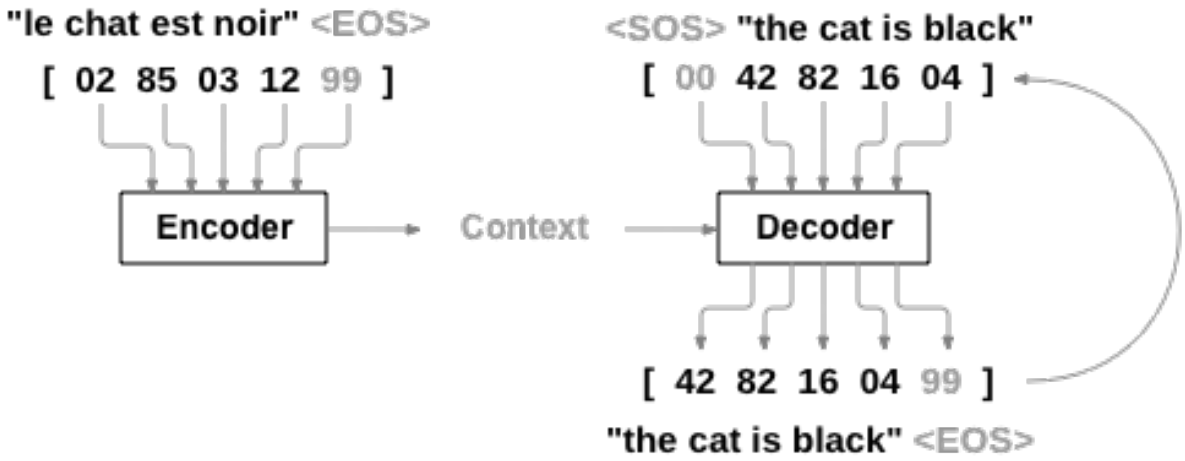
Out:

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10853 sentence pairs
Counting words...
Counted words:
fra 4489
eng 2925
['je vais te donner une lecon .', 'i m going to teach you a lesson .']
```

2.3.2 The Seq2Seq Model

A Recurrent Neural Network, or RNN, is a network that operates on a sequence and uses its own output as input for subsequent steps.

A Sequence to Sequence network, or seq2seq network, or Encoder Decoder network, is a model consisting of two RNNs called the encoder and decoder. The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence.



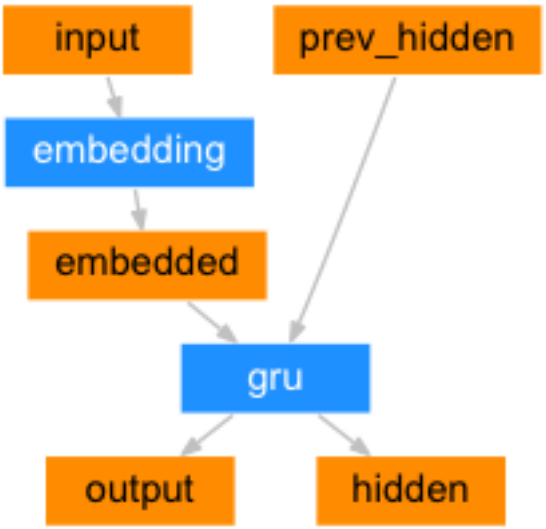
Unlike sequence prediction with a single RNN, where every input corresponds to an output, the seq2seq model frees us from sequence length and order, which makes it ideal for translation between two languages.

Consider the sentence "Je ne suis pas le chat noir" → "I am not the black cat". Most of the words in the input sentence have a direct translation in the output sentence, but are in slightly different orders, e.g. "chat noir" and "black cat". Because of the "ne/pas" construction there is also one more word in the input sentence. It would be difficult to produce a correct translation directly from the sequence of input words.

With a seq2seq model the encoder creates a single vector which, in the ideal case, encodes the "meaning" of the input sequence into a single vector — a single point in some N dimensional space of sentences.

The Encoder

The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word.



```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

The Decoder

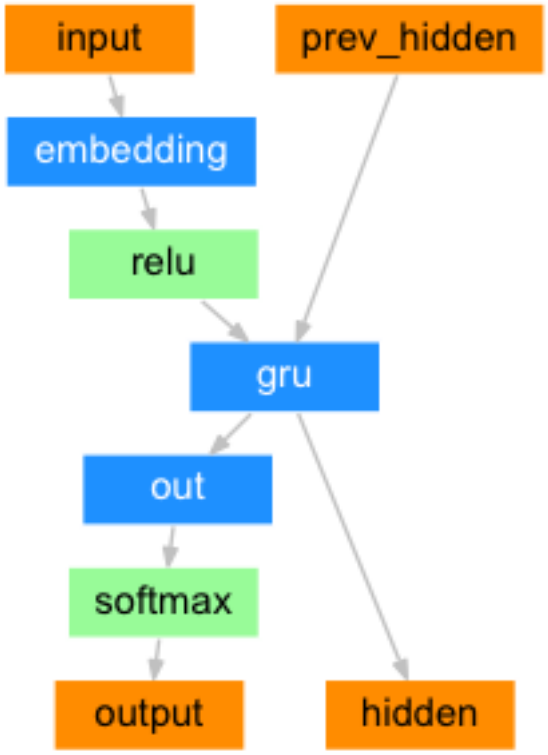
The decoder is another RNN that takes the encoder output vector(s) and outputs a sequence of words to create the translation.

Simple Decoder

In the simplest seq2seq decoder we use only last output of the encoder. This last output is sometimes called the *context vector* as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder.

At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string <SOS> token, and the first hidden state is the context vector (the encoder's last hidden state).

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
```




```
            return result.cuda()
        else:
            return result
```

I encourage you to train and observe the results of this model, but to save space we'll be going straight for the gold and introducing the Attention Mechanism.

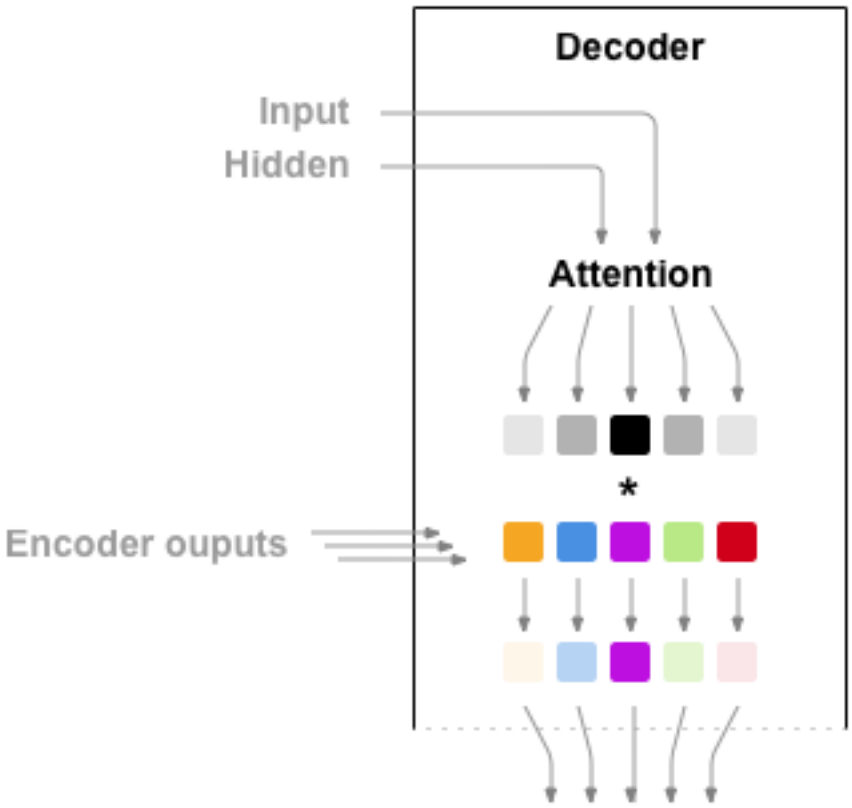
Attention Decoder

If only the context vector is passed betweeen the encoder and decoder, that single vector carries the burden of encoding the entire sentence.

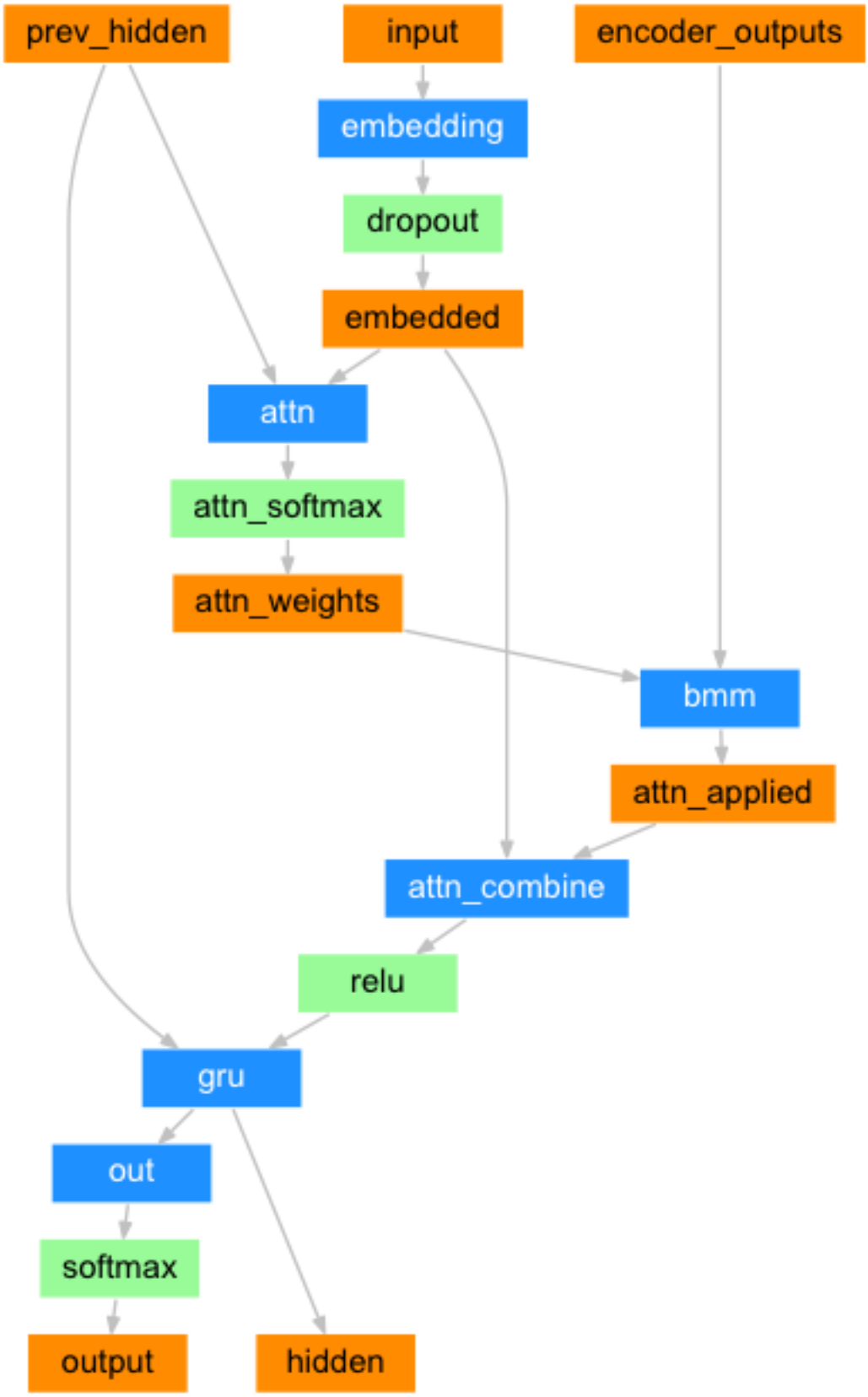
Attention allows the decoder network to "focus" on a different part of the encoder's outputs for every step of the decoder's own outputs. First we calculate a set of *attention weights*. These will be multiplied by the encoder output vectors to create a weighted combination. The result (called `attn_applied` in the code) should contain information about that specific part of the input sequence, and thus help the decoder choose the right output words.

Calculating the attention weights is done with another feed-forward layer `attn`, using the decoder's input and hidden state as inputs. Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to. Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few.

```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_
↪LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
```




```
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights
```


prev_hidden
input
encoder_outputs
embedding
dropout
embedded
attn
attn_softmax
attn_weights
bmm
attn_applied
attn_combine
relu
gru
out
softmax
output
hidden

```
    def initHidden(self):
        result = Variable(torch.zeros(1, 1, self.hidden_size))
        if use_cuda:
            return result.cuda()
        else:
            return result
```

Note: There are other forms of attention that work around the length limitation by using a relative position approach. Read about "local attention" in Effective Approaches to Attention-based Neural Machine Translation.

2.3.3 Training

Preparing Training Data

To train, for each pair we will need an input tensor (indexes of the words in the input sentence) and target tensor (indexes of the words in the target sentence). While creating these vectors we will append the EOS token to both sequences.

```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]


def variableFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    result = Variable(torch.LongTensor(indexes).view(-1, 1))
    if use_cuda:
        return result.cuda()
    else:
        return result


def variablesFromPair(pair):
    input_variable = variableFromSentence(input_lang, pair[0])
    target_variable = variableFromSentence(output_lang, pair[1])
    return (input_variable, target_variable)
```

Training the Model

To train we run the input sentence through the encoder, and keep track of every output and the latest hidden state. Then the decoder is given the <SOS> token as its first input, and the last hidden state of the encoder as its first hidden state.

"Teacher forcing" is the concept of using the real target outputs as each next input, instead of using the decoder's guess as the next input. Using teacher forcing causes it to converge faster but when the trained network is exploited, it may exhibit instability.

You can observe outputs of teacher-forced networks that read with coherent grammar but wander far from the correct translation - intuitively it has learned to represent the output grammar and can "pick up" the meaning once the teacher tells it the first few words, but it has not properly learned how to create the sentence from the translation in the first place.

Because of the freedom PyTorch's autograd gives us, we can randomly choose to use teacher forcing or not with a simple if statement. Turn `teacher_forcing_ratio` up to use more of it.

```
teacher_forcing_ratio = 0.5


def train(input_variable, target_variable, encoder, decoder, encoder_optimizer,␣
↪decoder_optimizer, criterion, max_length=MAX_LENGTH):
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_variable.size()[0]
    target_length = target_variable.size()[0]

    encoder_outputs = Variable(torch.zeros(max_length, encoder.hidden_size))
    encoder_outputs = encoder_outputs.cuda() if use_cuda else encoder_outputs

    loss = 0

    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_variable[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0][0]

    decoder_input = Variable(torch.LongTensor([[SOS_token]]))
    decoder_input = decoder_input.cuda() if use_cuda else decoder_input

    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_variable[di])
            decoder_input = target_variable[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.data.topk(1)
            ni = topi[0][0]

            decoder_input = Variable(torch.LongTensor([[ni]]))
            decoder_input = decoder_input.cuda() if use_cuda else decoder_input

            loss += criterion(decoder_output, target_variable[di])
            if ni == EOS_token:
                break

    loss.backward()

    encoder_optimizer.step()
```

```
    decoder_optimizer.step()

    return loss.data[0] / target_length
```

This is a helper function to print time elapsed and estimated time remaining given the current time and progress %.

```
import time
import math


def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)


def timeSince(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

The whole training process looks like this:

- Start a timer
- Initialize optimizers and criterion
- Create set of training pairs
- Start empty losses array for plotting

Then we call `train` many times and occasionally print the progress (% of examples, time so far, estimated time) and average loss.

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_
↪rate=0.01):
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate)
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate)
    training_pairs = [variablesFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_variable = training_pair[0]
        target_variable = training_pair[1]

        loss = train(input_variable, target_variable, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, criterion)
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
```

```
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters),
                                         iter, iter / n_iters * 100, print_loss_avg))

        if iter % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_losses.append(plot_loss_avg)
            plot_loss_total = 0

    showPlot(plot_losses)
```

Plotting results

Plotting is done with matplotlib, using the array of loss values `plot_losses` saved while training.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

2.3.4 Evaluation

Evaluation is mostly the same as training, but there are no targets so we simply feed the decoder's predictions back to itself for each step. Every time it predicts a word we add it to the output string, and if it predicts the EOS token we stop there. We also store the decoder's attention outputs for display later.

```
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    input_variable = variableFromSentence(input_lang, sentence)
    input_length = input_variable.size()[0]
    encoder_hidden = encoder.initHidden()

    encoder_outputs = Variable(torch.zeros(max_length, encoder.hidden_size))
    encoder_outputs = encoder_outputs.cuda() if use_cuda else encoder_outputs

    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(input_variable[ei],
                                                 encoder_hidden)
        encoder_outputs[ei] = encoder_outputs[ei] + encoder_output[0][0]

    decoder_input = Variable(torch.LongTensor([[SOS_token]]))  # SOS
    decoder_input = decoder_input.cuda() if use_cuda else decoder_input

    decoder_hidden = encoder_hidden

    decoded_words = []
    decoder_attentions = torch.zeros(max_length, max_length)
```

```
    for di in range(max_length):
        decoder_output, decoder_hidden, decoder_attention = decoder(
            decoder_input, decoder_hidden, encoder_outputs)
        decoder_attentions[di] = decoder_attention.data
        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0]
        if ni == EOS_token:
            decoded_words.append('<EOS>')
            break
        else:
            decoded_words.append(output_lang.index2word[ni])

        decoder_input = Variable(torch.LongTensor([[ni]]))
        decoder_input = decoder_input.cuda() if use_cuda else decoder_input

    return decoded_words, decoder_attentions[:di + 1]
```

We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements:

```
def evaluateRandomly(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

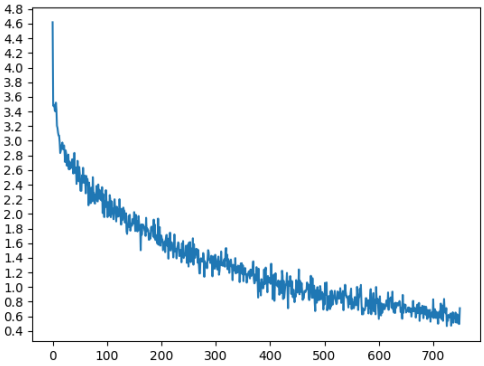
2.3.5 Training and Evaluating

With all these helper functions in place (it looks like extra work, but it makes it easier to run multiple experiments) we can actually initialize a network and start training.

Remember that the input sentences were heavily filtered. For this small dataset we can use relatively small networks of 256 hidden nodes and a single GRU layer. After about 40 minutes on a MacBook CPU we'll get some reasonable results.

Note: If you run this notebook you can train, interrupt the kernel, evaluate, and continue training later. Comment out the lines where the encoder and decoder are initialized and run `trainIters` again.

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1)


if use_cuda:
    encoder1 = encoder1.cuda()
    attn_decoder1 = attn_decoder1.cuda()

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

- 

Out:

```
11m 22s (- 159m 10s) (5000 6%) 2.9035
16m 22s (- 106m 25s) (10000 13%) 2.3034
20m 41s (- 82m 45s) (15000 20%) 1.9860
26m 33s (- 73m 1s) (20000 26%) 1.7745
31m 12s (- 62m 25s) (25000 33%) 1.5443
35m 41s (- 53m 32s) (30000 40%) 1.3865
40m 17s (- 46m 2s) (35000 46%) 1.3013
44m 46s (- 39m 10s) (40000 53%) 1.1276
49m 18s (- 32m 52s) (45000 60%) 1.0140
62m 27s (- 31m 13s) (50000 66%) 0.9333
67m 14s (- 24m 27s) (55000 73%) 0.8654
71m 39s (- 17m 54s) (60000 80%) 0.7873
84m 15s (- 12m 57s) (65000 86%) 0.7440
89m 37s (- 6m 24s) (70000 93%) 0.6706
94m 6s (- 0m 0s) (75000 100%) 0.6154
```

```
evaluateRandomly(encoder1, attn_decoder1)
```

Out:

```
> je suis crevee .
= i m exhausted .
< i m exhausted . <EOS>

> nous sommes differentes .
```

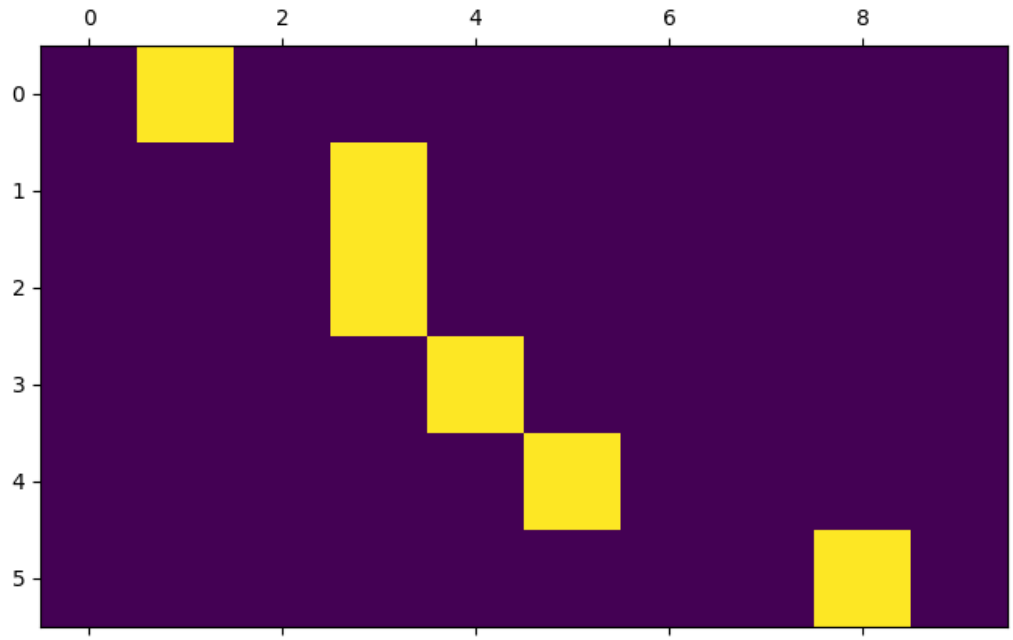
```
= we re different .
< we re different . <EOS>

> il est assis sur la chaise .
= he is sitting on the chair .
< he is sitting on the chair . <EOS>

> vous etes tres efficace .
= you re very efficient .
< you re very efficient . <EOS>

> je suis plus astucieux que vous .
= i m smarter than you .
< i m smarter than you . <EOS>

> elle est desireuse de visiter l europe .
= she is anxious to visit europe .
< she is anxious to apologize . <EOS>

> ils n ont pas froid aux yeux .
= they re brave .
< they re not prisoners . <EOS>

> il me degoute .
= i am disgusted with him .
< he is getting me by him . <EOS>

> vous etes fort attirant .
= you re very attractive .
< you re very attractive . <EOS>

> nous sommes tellement fieres de vous !
= we re so proud of you !
< we re so proud of you ! <EOS>
```

Visualizing Attention

A useful property of the attention mechanism is its highly interpretable outputs. Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step.

You could simply run `plt.matshow(attentions)` to see attention output displayed as a matrix, with the columns being input steps and rows being output steps:

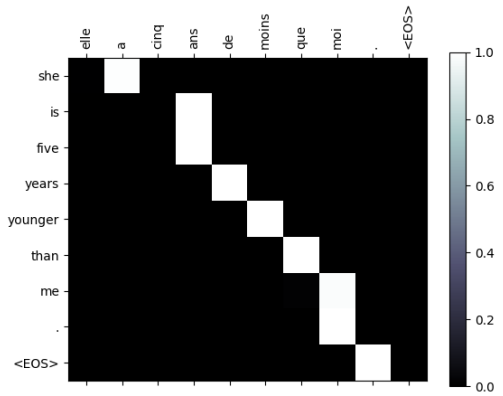
```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "je suis trop froid .")
plt.matshow(attentions.numpy())
```

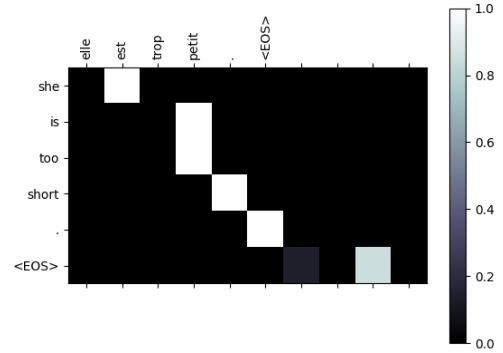


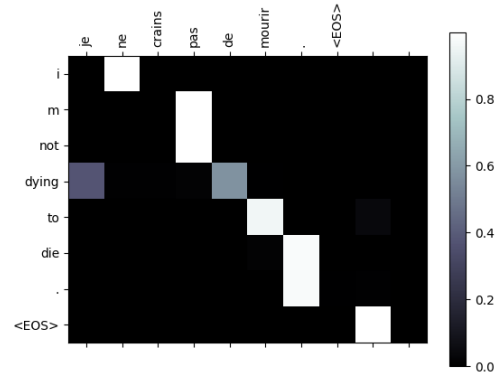
For a better viewing experience we will do the extra work of adding axes and labels:

```
def showAttention(input_sentence, output_words, attentions):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # Set up axes
    ax.set_xticklabels(['']  + input_sentence.split(' ') +
                       ['<EOS>'], rotation=90)
    ax.set_yticklabels(['']  + output_words)

    # Show label at every tick
    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    plt.show()


def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)


evaluateAndShowAttention("elle a cinq ans de moins que moi .")

evaluateAndShowAttention("elle est trop petit .")
```

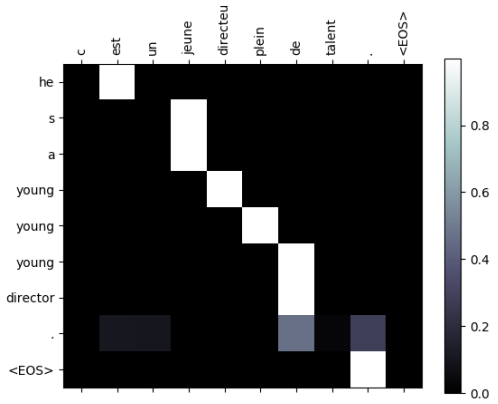
```
evaluateAndShowAttention("je ne crains pas de mourir .")

evaluateAndShowAttention("c est un jeune directeur plein de talent .")
```



- 

- 

-



-

Out:

```
input = elle a cinq ans de moins que moi .
output = she is five years younger than me . <EOS>
input = elle est trop petit .
output = she is too short . <EOS>
input = je ne crains pas de mourir .
output = i m not dying to die . <EOS>
input = c est un jeune directeur plein de talent .
output = he s a young young young director . <EOS>
```

2.3.6 Exercises

- Try with a different dataset
  - Another language pair
  - Human → Machine (e.g. IOT commands)
  - Chat → Response
  - Question → Answer
- Replace the embeddings with pre-trained word embeddings such as word2vec or GloVe
- Try with more layers, more hidden units, and more sentences. Compare the training time and results.
- If you use a translation file where pairs have two of the same phrase (`I am test \t I am test`), you can use this as an autoencoder. Try this:
  - Train as an autoencoder
  - Save only the Encoder network
  - Train a new Decoder for translation from there

Total running time of the script: (94 minutes 17.252 seconds)

`Download Python source code: seq2seq_translation_tutorial.py`

`Download Jupyter notebook: seq2seq_translation_tutorial.ipynb`

Generated by Sphinx-Gallery

2.4 Reinforcement Learning (DQN) tutorial

Author: Adam Paszke

This tutorial shows how to use PyTorch to train a Deep Q Learning (DQN) agent on the CartPole-v0 task from the OpenAI Gym.

Task

The agent has to decide between two actions - moving the cart left or right - so that the pole attached to it stays upright. You can find an official leaderboard with various algorithms and visualizations at the Gym website.

Fig. 2.1: cartpole

As the agent observes the current state of the environment and chooses an action, the environment *transitions* to a new state, and also returns a reward that indicates the consequences of the action. In this task, the environment terminates if the pole falls over too far.

The CartPole task is designed so that the inputs to the agent are 4 real values representing the environment state (position, velocity, etc.). However, neural networks can solve the task purely by looking at the scene, so we'll use a patch of the screen centered on the cart as an input. Because of this, our results aren't directly comparable to the ones from the official leaderboard - our task is much harder. Unfortunately this does slow down the training, because we have to render all the frames.

Strictly speaking, we will present the state as the difference between the current screen patch and the previous one. This will allow the agent to take the velocity of the pole into account from one image.

Packages

First, let's import needed packages. Firstly, we need gym for the environment (Install using *pip install gym*). We'll also use the following from PyTorch:

- neural networks (`torch.nn`)
- optimization (`torch.optim`)
- automatic differentiation (`torch.autograd`)
- utilities for vision tasks (`torchvision` - a separate package).

```
import gym
import math
import random
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple
from itertools import count
from copy import deepcopy
from PIL import Image

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import torchvision.transforms as T
```

```
env = gym.make('CartPole-v0').unwrapped

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

# if gpu is to be used
use_cuda = torch.cuda.is_available()
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor
```

2.4.1 Replay Memory

We'll be using experience replay memory for training our DQN. It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.

For this, we're going to need two classses:

- `Transition` - a named tuple representing a single transition in our environment
- `ReplayMemory` - a cyclic buffer of bounded size that holds the transitions observed recently. It also implements a `.sample()` method for selecting a random batch of transitions for training.

```
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

Now, let's define our model. But first, let quickly recap what a DQN is.

2.4.2 DQN algorithm

Our environment is deterministic, so all equations presented here are also formulated deterministically for the sake of simplicity. In the reinforcement learning literature, they would also contain expectations over stochastic transitions in the environment.

Our aim will be to train a policy that tries to maximize the discounted, cumulative reward $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$, where R_{t_0} is also known as the *return*. The discount, γ, should be a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future that it can be fairly confident about.

The main idea behind Q-learning is that if we had a function $Q^* : State \times Action \to \mathbb{R}$, that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s, a)$$

However, we don't know everything about the world, so we don't have access to Q^*. But, since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*.

For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation:

$$Q^{\pi}(s, a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

The difference between the two sides of the equality is known as the temporal difference error, δ:

$$\delta = Q(s, a) - (r + \gamma \max_a Q(s', a))$$

To minimise this error, we will use the Huber loss. The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large - this makes it more robust to outliers when the estimates of Q are very noisy. We calculate this over a batch of transitions, B, sampled from the replay memory:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r)\ \in\ B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \le 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

Q-network

Our model will be a convolutional neural network that takes in the difference between the current and previous screen patches. It has two outputs, representing $Q(s, \text{left})$ and $Q(s, \text{right})$ (where s is the input to the network). In effect, the network is trying to predict the *quality* of taking each action given the current input.

```
class DQN(nn.Module):

    def __init__(self):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=2)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=2)
        self.bn3 = nn.BatchNorm2d(32)
        self.head = nn.Linear(448, 2)
```

```
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        return self.head(x.view(x.size(0), -1))
```

Input extraction

The code below are utilities for extracting and processing rendered images from the environment. It uses the `torchvision` package, which makes it easy to compose image transforms. Once you run the cell it will display an example patch that it extracted.

```
resize = T.Compose([T.ToPILImage(),
                    T.Scale(40, interpolation=Image.CUBIC),
                    T.ToTensor()])

# This is based on the code from gym.
screen_width = 600


def get_cart_location():
    world_width = env.x_threshold * 2
    scale = screen_width / world_width
    return int(env.state[0] * scale + screen_width / 2.0)  # MIDDLE OF CART


def get_screen():
    screen = env.render(mode='rgb_array').transpose(
        (2, 0, 1))  # transpose into torch order (CHW)
    # Strip off the top and bottom of the screen
    screen = screen[:, 160:320]
    view_width = 320
    cart_location = get_cart_location()
    if cart_location < view_width // 2:
        slice_range = slice(view_width)
    elif cart_location > (screen_width - view_width // 2):
        slice_range = slice(-view_width, None)
    else:
        slice_range = slice(cart_location - view_width // 2,
                            cart_location + view_width // 2)
    # Strip off the edges, so that we have a square image centered on a cart
    screen = screen[:, :, slice_range]
    # Convert to float, rescare, convert to torch tensor
    # (this doesn't require a copy)
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255
    screen = torch.from_numpy(screen)
    # Resize, and add a batch dimension (BCHW)
    return resize(screen).unsqueeze(0).type(Tensor)

env.reset()
plt.figure()
plt.imshow(get_screen().cpu().squeeze(0).permute(1, 2, 0).numpy(),
           interpolation='none')
plt.title('Example extracted screen')
plt.show()
```

2.4.3 Training

Hyperparameters and utilities

This cell instantiates our model and its optimizer, and defines some utilities:

- `Variable` - this is a simple wrapper around `torch.autograd.Variable` that will automatically send the data to the GPU every time we construct a Variable.
- `select_action` - will select an action accordingly to an epsilon greedy policy. Simply put, we'll sometimes use our model for choosing the action, and sometimes we'll just sample one uniformly. The probability of choosing a random action will start at `EPS_START` and will decay exponentially towards `EPS_END`. `EPS_DECAY` controls the rate of the decay.
- `plot_durations` - a helper for plotting the durations of episodes, along with an average over the last 100 episodes (the measure used in the official evaluations). The plot will be underneath the cell containing the main training loop, and will update after every episode.

```
BATCH_SIZE = 128
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200

model = DQN()

if use_cuda:
    model.cuda()

optimizer = optim.RMSprop(model.parameters())
memory = ReplayMemory(10000)


steps_done = 0


def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        return model(
            Variable(state, volatile=True).type(FloatTensor)).data.max(1)[1].view(1,␣
↪1)
    else:
        return LongTensor([[random.randrange(2)]])


episode_durations = []


def plot_durations():
    plt.figure(2)
    plt.clf()
    durations_t = torch.FloatTensor(episode_durations)
    plt.title('Training...')
    plt.xlabel('Episode')
```

```
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        display.clear_output(wait=True)
        display.display(plt.gcf())
```

Training loop

Finally, the code for training our model.

Here, you can find an `optimize_model` function that performs a single step of the optimization. It first samples a batch, concatenates all the tensors into a single one, computes $Q(s_t, a_t)$ and $V(s_{t+1}) = \max_a Q(s_{t+1}, a)$, and combines them into our loss. By defition we set $V(s) = 0$ if s is a terminal state.

```
last_sync = 0


def optimize_model():
    global last_sync
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see http://stackoverflow.com/a/19343/3343043 for
    # detailed explanation).
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = ByteTensor(tuple(map(lambda s: s is not None,
                                          batch.next_state)))

    # We don't want to backprop through the expected action values and volatile
    # will save us on temporarily changing the model parameters'
    # requires_grad to False!
    non_final_next_states = Variable(torch.cat([s for s in batch.next_state
                                                if s is not None]),
                                     volatile=True)
    state_batch = Variable(torch.cat(batch.state))
    action_batch = Variable(torch.cat(batch.action))
    reward_batch = Variable(torch.cat(batch.reward))

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken
    state_action_values = model(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    next_state_values = Variable(torch.zeros(BATCH_SIZE).type(Tensor))
    next_state_values[non_final_mask] = model(non_final_next_states).max(1)[0]
    # Now, we don't want to mess up the loss with a volatile flag, so let's
    # clear it. After this, we'll just end up with a Variable that has
    # requires_grad=False
```

```
    next_state_values.volatile = False
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    loss = F.smooth_l1_loss(state_action_values, expected_state_action_values)

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    for param in model.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

Below, you can find the main training loop. At the beginning we reset the environment and initialize the `state` variable. Then, we sample an action, execute it, observe the next screen and the reward (always 1), and optimize our model once. When the episode ends (our model fails), we restart the loop.

Below, *num_episodes* is set small. You should download the notebook and run lot more epsiodes.

```
num_episodes = 10
for i_episode in range(num_episodes):
    # Initialize the environment and state
    env.reset()
    last_screen = get_screen()
    current_screen = get_screen()
    state = current_screen - last_screen
    for t in count():
        # Select and perform an action
        action = select_action(state)
        _, reward, done, _ = env.step(action[0, 0])
        reward = Tensor([reward])

        # Observe new state
        last_screen = current_screen
        current_screen = get_screen()
        if not done:
            next_state = current_screen - last_screen
        else:
            next_state = None

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the target network)
        optimize_model()
        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break

print('Complete')
env.render(close=True)
env.close()
plt.ioff()
```

```
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

2.5 Writing Distributed Applications with PyTorch

Author: Séb Arnold

In this short tutorial, we will be going over the distributed package of PyTorch. We'll see how to set up the distributed setting, use the different communication strategies, and go over some the internals of the package.

2.5.1 Setup

The distributed package included in PyTorch (i.e., `torch.distributed`) enables researchers and practitioners to easily parallelize their computations across processes and clusters of machines. To do so, it leverages the messaging passing semantics allowing each process to communicate data to any of the other processes. As opposed to the multiprocessing (`torch.multiprocessing`) package, processes can use different communication backends and are not restricted to being executed on the same machine.

In order to get started we need the ability to run multiple processes simultaneously. If you have access to compute cluster you should check with your local sysadmin or use your favorite coordination tool. (e.g., pdsh, clustershell, or others) For the purpose of this tutorial, we will use a single machine and fork multiple processes using the following template.

```
"""run.py:"""
#!/usr/bin/env python
import os
import torch
import torch.distributed as dist
from torch.multiprocessing import Process

def run(rank, size):
    """ Distributed function to be implemented later. """
    pass

def init_processes(rank, size, fn, backend='tcp'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)


if __name__ == "__main__":
    size = 2
    processes = []
    for rank in range(size):
        p = Process(target=init_processes, args=(rank, size, run))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()
```

The above script spawns two processes who will each setup the distributed environment, initialize the process group (`dist.init_process_group`), and finally execute the given `run` function.

Let's have a look at the `init_processes` function. It ensures that every process will be able to coordinate through a master, using the same ip address and port. Note that we used the TCP backend, but we could have used MPI or Gloo instead. (c.f. *Section 5.1*) We will go over the magic happening in `dist.init_process_group` at the end of this tutorial, but it essentially allows processes to communicate with each other by sharing their locations.

2.5.2 Point-to-Point Communication

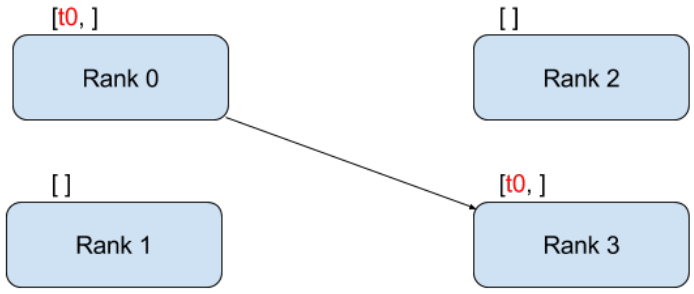


Fig. 2.2: Send and Recv

A transfer of data from one process to another is called a point-to-point communication. These are achieved through the `send` and `recv` functions or their *immediate* counter-parts, `isend` and `irecv`.

```
"""Blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
        dist.send(tensor=tensor, dst=1)
    else:
        # Receive tensor from process 0
        dist.recv(tensor=tensor, src=0)
    print('Rank ', rank, ' has data ', tensor[0])
```

In the above example, both processes start with a zero tensor, then process 0 increments the tensor and sends it to process 1 so that they both end up with 1.0. Notice that process 1 needs to allocate memory in order to store the data it will receive.

Also notice that `send`/`recv` are **blocking**: both processes stop until the communication is completed. On the other hand immediates are **non-blocking**; the script continues its execution and the methods return a `DistributedRequest` object upon which we can choose to `wait()`.

```
"""Non-blocking point-to-point communication."""

def run(rank, size):
    tensor = torch.zeros(1)
    req = None
    if rank == 0:
        tensor += 1
        # Send the tensor to process 1
```

```
        req = dist.isend(tensor=tensor, dst=1)
        print('Rank 0 started sending')
    else:
        # Receive tensor from process 0
        req = dist.irecv(tensor=tensor, src=0)
        print('Rank 1 started receiving')
    req.wait()
    print('Rank ', rank, ' has data ', tensor[0])
```

When using immediates we have to be careful about with our usage of the sent and received tensors. Since we do not know when the data will be communicated to the other process, we should not modify the sent tensor nor access the received tensor before `req.wait()` has completed. In other words,

- writing to `tensor` after `dist.isend()` will result in undefined behaviour.
- reading from `tensor` after `dist.irecv()` will result in undefined behaviour.

However, after `req.wait()` has been executed we are guaranteed that the communication took place, and that the value stored in `tensor[0]` is 1.0.

Point-to-point communication is useful when we want a fine-grained control over the communication of our processes. They can be used to implement fancy algorithms, such as the one used in Baidu's DeepSpeech or Facebook's large-scale experiments.(c.f. *Section 4.1*)

2.5.3 Collective Communication

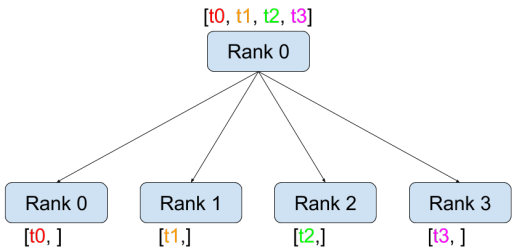


Fig. 2.3: Scatter

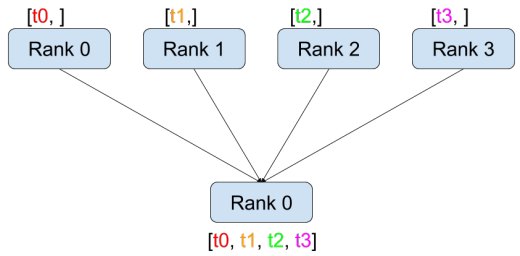


Fig. 2.4: Gather

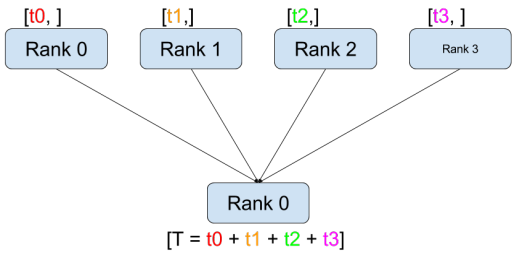


Fig. 2.5: Reduce

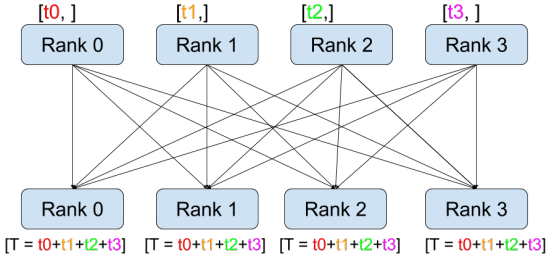


Fig. 2.6: All-Reduce

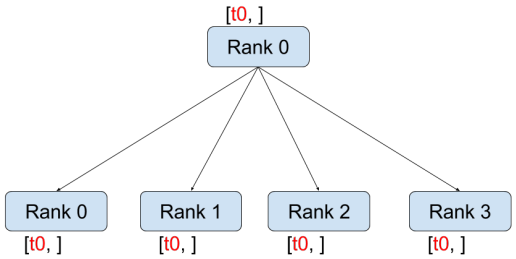


Fig. 2.7: Broadcast

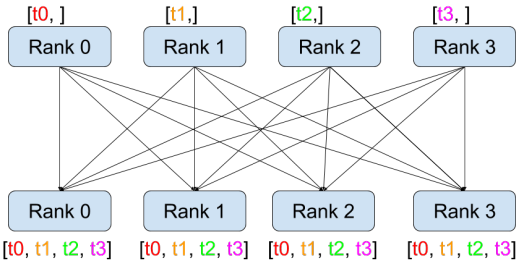


Fig. 2.8: All-Gather

As opposed to point-to-point communcation, collectives allow for communication patterns across all processes in a **group**. A group is a subset of all our processes. To create a group, we can pass a list of ranks to `dist.new_group(group)`. By default, collectives are executed on the all processes, also known as the **world**. For example, in order to obtain the sum of all tensors at all processes, we can use the `dist.all_reduce(tensor, op, group)` collective.

```
""" All-Reduce example."""
def run(rank, size):
    """ Simple point-to-point communication. """
    group = dist.new_group([0, 1])
    tensor = torch.ones(1)
    dist.all_reduce(tensor, op=dist.reduce_op.SUM, group=group)
    print('Rank ', rank, ' has data ', tensor[0])
```

Since we want the sum of all tensors in the group, we use `dist.reduce_op.SUM` as the reduce operator. Generally speaking, any commutative mathematical operation can be used as an operator. Out-of-the-box, PyTorch comes with 4 such operators, all working at the element-wise level:

- `dist.reduce_op.SUM`,
- `dist.reduce_op.PRODUCT`,
- `dist.reduce_op.MAX`,

- `dist.reduce_op.MIN`.

In addition to `dist.all_reduce(tensor, op, group)`, there are a total of 6 collectives currently implemented in PyTorch.

- `dist.broadcast(tensor, src, group)`: Copies `tensor` from `src` to all other processes.
- `dist.reduce(tensor, dst, op, group)`: Applies `op` to all `tensor` and stores the result in `dst`.
- `dist.all_reduce(tensor, op, group)`: Same as reduce, but the result is stored in all processes.
- `dist.scatter(tensor, src, scatter_list, group)`: Copies the i^{th} tensor `scatter_list[i]` to the i^{th} process.
- `dist.gather(tensor, dst, gather_list, group)`: Copies `tensor` from all processes in `dst`.
- `dist.all_gather(tensor_list, tensor, group)`: Copies `tensor` from all processes to `tensor_list`, on all processes.

2.5.4 Distributed Training

Note: You can find the example script of this section in this GitHub repository.

Now that we understand how the distributed module works, let us write something useful with it. Our goal will be to replicate the functionality of DistributedDataParallel. Of course, this will be a didactic example and in a real-world situtation you should use the official, well-tested and well-optimized version linked above.

Quite simply we want to implement a distributed version of stochastic gradient descent. Our script will let all processes compute the gradients of their model on their batch of data and then average their gradients. In order to ensure similar convergence results when changing the number of processes, we will first have to partition our dataset. (You could also use tnt.dataset.SplitDataset, instead of the snippet below.)

```
""" Dataset partitioning helper """
class Partition(object):

    def __init__(self, data, index):
        self.data = data
        self.index = index

    def __len__(self):
        return len(self.index)

    def __getitem__(self, index):
        data_idx = self.index[index]
        return self.data[data_idx]


class DataPartitioner(object):

    def __init__(self, data, sizes=[0.7, 0.2, 0.1], seed=1234):
        self.data = data
        self.partitions = []
        rng = Random()
        rng.seed(seed)
        data_len = len(data)
        indexes = [x for x in range(0, data_len)]
        rng.shuffle(indexes)

        for frac in sizes:
            part_len = int(frac * data_len)
```

```
            self.partitions.append(indexes[0:part_len])
            indexes = indexes[part_len:]

    def use(self, partition):
        return Partition(self.data, self.partitions[partition])
```

With the above snippet, we can now simply partition any dataset using the following few lines:

```
""" Partitioning MNIST """
def partition_dataset():
    dataset = datasets.MNIST('./data', train=True, download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.1307,), (0.3081,))
                             ]))
    size = dist.get_world_size()
    bsz = 128 / float(size)
    partition_sizes = [1.0 / size for _ in range(size)]
    partition = DataPartitioner(dataset, partition_sizes)
    partition = partition.use(dist.get_rank())
    train_set = torch.utils.data.DataLoader(partition,
                                            batch_size=bsz,
                                            shuffle=True)
    return train_set, bsz
```

Assuming we have 2 replicas, then each process will have a `train_set` of 60000 / 2 = 30000 samples. We also divide the batch size by the number of replicas in order to maintain the *overall* batch size of 128.

We can now write our usual forward-backward-optimize training code, and add a function call to average the gradients of our models. (The following is largely inspired from the official PyTorch MNIST example.)

```
""" Distributed Synchronous SGD Example """
def run(rank, size):
        torch.manual_seed(1234)
        train_set, bsz = partition_dataset()
        model = Net()
        optimizer = optim.SGD(model.parameters(),
                              lr=0.01, momentum=0.5)

        num_batches = ceil(len(train_set.dataset) / float(bsz))
        for epoch in range(10):
            epoch_loss = 0.0
            for data, target in train_set:
                data, target = Variable(data), Variable(target)
                optimizer.zero_grad()
                output = model(data)
                loss = F.nll_loss(output, target)
                epoch_loss += loss.data[0]
                loss.backward()
                average_gradients(model)
                optimizer.step()
            print('Rank ', dist.get_rank(), ', epoch ',
                  epoch, ': ', epoch_loss / num_batches)
```

It remains to implement the `average_gradients(model)` function, which simply takes in a model and averages its gradients across the whole world.

```
""" Gradient averaging. """
def average_gradients(model):
    size = float(dist.get_world_size())
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.reduce_op.SUM)
        param.grad.data /= size
```

Et voilà! We successfully implemented distributed synchronous SGD and could train any model on a large computer cluster.

Note: While the last sentence is *technically* true, there are a lot more tricks required to implement a production-level implementation of synchronous SGD. Again, use what has been tested and optimized.

Our Own Ring-Allreduce

As an additional challenge, imagine that we wanted to implement DeepSpeech's efficient ring allreduce. This is fairly easily implemented using point-to-point collectives.

```
""" Implementation of a ring-reduce with addition. """
def allreduce(send, recv):
    rank = dist.get_rank()
    size = dist.get_world_size()
    send_buff = th.zeros(send.size())
    recv_buff = th.zeros(send.size())
    accum = th.zeros(send.size())
    accum[:] = send[:]

    left = ((rank - 1) + size) % size
    right = (rank + 1) % size

    for i in range(size - 1):
        if i % 2 == 0:
            # Send send_buff
            send_req = dist.isend(send_buff, right)
            dist.recv(recv_buff, left)
            accum[:] += recv[:]
        else:
            # Send recv_buff
            send_req = dist.isend(recv_buff, right)
            dist.recv(send_buff, left)
            accum[:] += send[:]
        send_req.wait()
    recv[:] = accum[:]
```

In the above script, the `allreduce(send, recv)` function has a slightly different signature than the ones in PyTorch. It takes a `recv` tensor and will store the sum of all `send` tensors in it. As an exercise left to the reader, there is still one difference between our version and the one in DeepSpeech: their implementation divide the gradient tensor into *chunks*, so as to optimially utilize the communication bandwidth. (Hint: toch.chunk)

2.5.5 Advanced Topics

We are now ready to discover some of the more advanced functionalities of `torch.distributed`. Since there is a lot to cover, this section is divided into two subsections:

1. Communication Backends: where we learn how to use MPI and Gloo for GPU-GPU communication.

2. Initialization Methods: where we understand how to best setup the initial coordination phase in `dist.init_process_group()`.

Communication Backends

One of the most elegant aspects of `torch.distributed` is its ability to abstract and build on top of different backends. As mentioned before, there are currently three backends implemented in PyTorch: TCP, MPI, and Gloo. They each have different specifications and tradeoffs, depending on the desired use-case. A comparative table of supported functions can be found here.

TCP Backend

So far we have made extensive usage of the TCP backend. It is quite handy as a development platform, as it is guaranteed to work on most machines and operating systems. It also supports all point-to-point and collective functions on CPU. However, there is no support for GPUs and its communication routines are not as optimized as the MPI one.

Gloo Backend

The Gloo backend provides an optimized implementation of *collective* communication procedures, both for CPUs and GPUs. It particularly shines on GPUs as it can perform communication without transferring data to the CPU's memory using GPUDirect. It is also capable of using NCCL to perform fast intra-node communication and implements its own algorithms for inter-node routines.

Since version 0.2.0, the Gloo backend is automatically included with the pre-compiled binaries of PyTorch. As you have surely noticed, our distributed SGD example does not work if you put `model` on the GPU. Let's fix it by first replacing `backend='gloo'` in `init_processes(rank, size, fn, backend='tcp')`. At this point, the script will still run on CPU but uses the Gloo backend behind the scenes. In order to use multiple GPUs, let us also do the following modifications:

0. `init_processes(rank, size, fn, backend='tcp')` → `init_processes(rank, size, fn, backend='gloo')`
1. `model = Net()` → `model = Net().cuda(rank)`
2. `data, target = Variable(data), Variable(target)` → `data, target = Variable(data.cuda(rank)), Variable(target.cuda(rank))`

With the above modifications, our model is now training on two GPUs and you can monitor their utilization with `watch nvidia-smi`.

MPI Backend

The Message Passing Interface (MPI) is a standardized tool from the field of high-performance computing. It allows to do point-to-point and collective communications and was the main inspiration for the API of `torch.distributed`. Several implementations of MPI exist (e.g. Open-MPI, MVAPICH2, Intel MPI) each optimized for different purposes. The advantage of using the MPI backend lies in MPI's wide availability - and high-level of optimization - on large computer clusters. Some recent implementations are also able to take advantage of CUDA IPC and GPU Direct technologies in order to avoid memory copies through the CPU.

Unfortunately, PyTorch's binaries can not include an MPI implementation and we'll have to recompile it by hand. Fortunately, this process is fairly simple given that upon compilation, PyTorch will look *by itself* for an available MPI implementation. The following steps install the MPI backend, by installing PyTorch from sources.

1. Create and activate your Anaconda environment, install all the pre-requisites following the guide, but do **not** run `python setup.py install` yet.
2. Choose and install your favorite MPI implementation. Note that enabling CUDA-aware MPI might require some additional steps. In our case, we'll stick to Open-MPI *without* GPU support: `conda install -c conda-forge openmpi`
3. Now, go to your cloned PyTorch repo and execute `python setup.py install`.

In order to test our newly installed backend, a few modifications are required.

1. Replace the content under `if __name__ == '__main__':` with `init_processes(0, 0, run, backend='mpi')`.
2. Run `mpirun -n 4 python myscript.py`.

The reason for these changes is that MPI needs to create its own environment before spawning the processes. MPI will also spawn its own processes and perform the handshake described in *Initialization Methods*, making the `rank`and `size` arguments of `init_process_group` superfluous. This is actually quite powerful as you can pass additional arguments to `mpirun` in order to tailor computational resources for each process. (Things like number of cores per process, hand-assigning machines to specific ranks, and some more) Doing so, you should obtain the same familiar output as with the other communication backends.

Initialization Methods

To finish this tutorial, let's talk about the very first function we called: `dist.init_process_group(backend, init_method)`. In particular, we will go over the different initialization methods which are responsible for the initial coordination step between each process. Those methods allow you to define how this coordination is done. Depending on your hardware setup, one of these methods should be naturally more suitable than the others. In addition to the following sections, you should also have a look at the official documentation.

Before diving into the initialization methods, let's have a quick look at what happens behind `init_process_group` from the C/C++ perspective.

1. First, the arguments are parsed and validated.
2. The backend is resolved via the `name2channel.at()` function. A `Channel` class is returned, and will be used to perform the data transmission.
3. The GIL is dropped, and `THDProcessGroupInit()` is called. This instantiates the channel and adds the address of the master node.
4. The process with rank 0 will execute the `master` procedure, while all other ranks will be `workers`.
5. The master
   (a) Creates sockets for all workers.
   (b) Waits for all workers to connect.
   (c) Sends them information about the location of the other processes.
6. Each worker
   (a) Creates a socket to the master.
   (b) Sends their own location information.
   (c) Receives information about the other workers.
   (d) Opens a socket and handshakes with all other workers.
7. The initialization is done, and everyone is connected to everyone.

Environment Variable

We have been using the environment variable initialization method throughout this tutorial. By setting the following four environment variables on all machines, all processes will be able to properly connect to the master, obtain information about the other processes, and finally handshake with them.

- `MASTER_PORT`: A free port on the machine that will host the process with rank 0.
- `MASTER_ADDR`: IP address of the machine that will host the process with rank 0.

- `WORLD_SIZE`: The total number of processes, so that the master knows how many workers to wait for.
- `RANK`: Rank of each process, so they will know whether it is the master of a worker.

Shared File System

The shared filesystem requires all processes to have access to a shared file system, and will coordinate them through a shared file. This means that each process will open the file, write its information, and wait until everybody did so. After what all required information will be readily available to all processes. In order to avoid race conditions, the file system must support locking through fcntl. Note that you can specify ranks manually or let the processes figure it out by themselves. Be defining a unique `groupname` per job you can use the same file path for multiple jobs and safely avoid collision.

```
dist.init_process_group(init_method='file:///mnt/nfs/sharedfile', world_size=4,
                        group_name='mygroup')
```

TCP Init & Multicast

Initializing via TCP can be achieved in two different ways:

1. By providing the IP address of the process with rank 0 and the world size.
2. By providing *any* valid IP multicast address and the world size.

In the first case, all workers will be able to connect to the process with rank 0 and follow the procedure described above.

```
dist.init_process_group(init_method='tcp://10.1.1.20:23456', rank=args.rank, world_
↪size=4)
```

In the second case, the multicast address specifies the group of nodes who might potentially be active and the coordination can be handled by allowing each process to have an initial handshake before following the above procedure. In addition TCP multicast initialization also supports a `group_name` argument (as with the shared file method) allowing multiple jobs to be scheduled on the same cluster.

```
dist.init_process_group(init_method='tcp://
↪[ff15:1e18:5d4c:4cf0:d02d:b659:53ba:b0a7]:23456',
                        world_size=4)
```

Acknowledgements

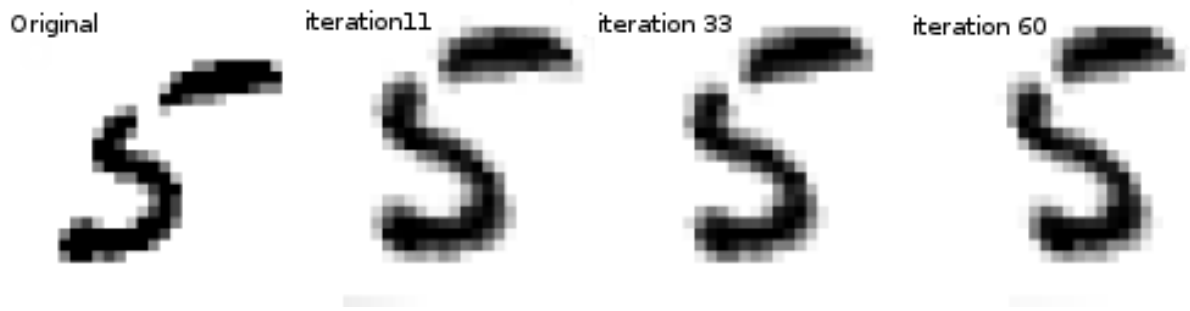
I'd like to thank the PyTorch developers for doing such a good job on their implementation, documentation, and tests. When the code was unclear, I could always count on the docs or the tests to find an answer. In particular, I'd like to thank Soumith Chintala, Adam Paszke, and Natalia Gimelshein for providing insightful comments and answering questions on early drafts.

2.6 Spatial Transformer Networks Tutorial

Author: Ghassen HAMROUNI

In this tutorial, you will learn how to augment your network using a visual attention mechanism called spatial transformer networks. You can read more about the spatial transformer networks in the DeepMind paper

Spatial transformer networks are a generalization of differentiable attention to any spatial transformation. Spatial transformer networks (STN for short) allow a neural network to learn how to perform spatial transformations on the input image in order to enhance the geometric invariance of the model. For example, it can crop a region of interest, scale and correct the orientation of an image. It can be a useful mechanism because CNNs are not invariant to rotation and scale and more general affine transformations.



One of the best things about STN is the ability to simply plug it into any existing CNN with very little modification.

```
# License: BSD
# Author: Ghassen Hamrouni

from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
from torch.autograd import Variable
import matplotlib.pyplot as plt
import numpy as np

plt.ion()   # interactive mode
```

2.6.1 Loading the data

In this post we experiment with the classic MNIST dataset. Using a standard convolutional network augmented with a spatial transformer network.

```
use_cuda = torch.cuda.is_available()

# Training dataset
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='.', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])), batch_size=64, shuffle=True, num_workers=4)
# Test dataset
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST(root='.', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])), batch_size=64, shuffle=True, num_workers=4)
```

Out:

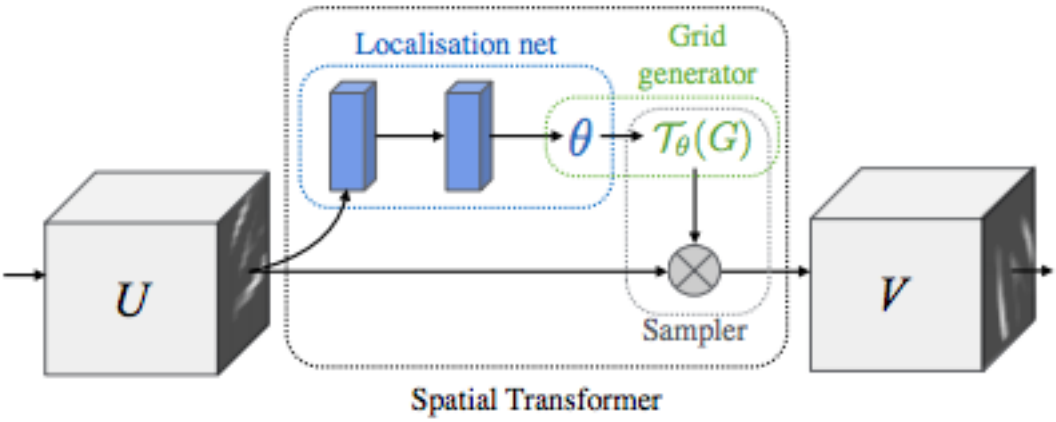
```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

2.6.2 Depicting spatial transformer networks

Spatial transformer networks boils down to three main components :

- The localization network is a regular CNN which regresses the transformation parameters. The transformation is never learned explicitly from this dataset, instead the network learns automatically the spatial transformations that enhances the global accuracy.
- The grid generator generates a grid of coordinates in the input image corresponding to each pixel from the output image.
- The sampler uses the parameters of the transformation and applies it to the input image.



Note: We need the latest version of PyTorch that contains affine_grid and grid_sample modules.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

        # Spatial transformer localization-network
        self.localization = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=7),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True),
            nn.Conv2d(8, 10, kernel_size=5),
            nn.MaxPool2d(2, stride=2),
            nn.ReLU(True)
        )
```

```
        # Regressor for the 3 * 2 affine matrix
        self.fc_loc = nn.Sequential(
            nn.Linear(10 * 3 * 3, 32),
            nn.ReLU(True),
            nn.Linear(32, 3 * 2)
        )

        # Initialize the weights/bias with identity transformation
        self.fc_loc[2].weight.data.fill_(0)
        self.fc_loc[2].bias.data = torch.FloatTensor([1, 0, 0, 0, 1, 0])

    # Spatial transformer network forward function
    def stn(self, x):
        xs = self.localization(x)
        xs = xs.view(-1, 10 * 3 * 3)
        theta = self.fc_loc(xs)
        theta = theta.view(-1, 2, 3)

        grid = F.affine_grid(theta, x.size())
        x = F.grid_sample(x, grid)

        return x

    def forward(self, x):
        # transform the input
        x = self.stn(x)

        # Perform the usual forward pass
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)


model = Net()
if use_cuda:
    model.cuda()
```

2.6.3 Training the model

Now, let's use the SGD algorithm to train the model. The network is learning the classification task in a supervised way. In the same time the model is learning STN automatically in an end-to-end fashion.

```
optimizer = optim.SGD(model.parameters(), lr=0.01)


def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if use_cuda:
            data, target = data.cuda(), target.cuda()

        data, target = Variable(data), Variable(target)
```

```
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 500 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
#
# A simple test procedure to measure STN the performances on MNIST.
#


def test():
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)

        # sum up batch loss
        test_loss += F.nll_loss(output, target, size_average=False).data[0]
        # get the index of the max log-probability
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'
          .format(test_loss, correct, len(test_loader.dataset),
                  100. * correct / len(test_loader.dataset)))
```

2.6.4 Visualizing the STN results

Now, we will inspect the results of our learned visual attention mechanism.

We define a small helper function in order to visualize the transformations while training.

```
def convert_image_np(inp):
    """Convert a Tensor to numpy image."""
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    return inp

# We want to visualize the output of the spatial transformers layer
# after the training, we visualize a batch of input images and
# the corresponding transformed batch using STN.


def visualize_stn():
    # Get a batch of training data
```

```
    data, _ = next(iter(test_loader))
    data = Variable(data, volatile=True)

    if use_cuda:
        data = data.cuda()

    input_tensor = data.cpu().data
    transformed_input_tensor = model.stn(data).cpu().data

    in_grid = convert_image_np(
        torchvision.utils.make_grid(input_tensor))

    out_grid = convert_image_np(
        torchvision.utils.make_grid(transformed_input_tensor))

    # Plot the results side-by-side
    f, axarr = plt.subplots(1, 2)
    axarr[0].imshow(in_grid)
    axarr[0].set_title('Dataset Images')

    axarr[1].imshow(out_grid)
    axarr[1].set_title('Transformed Images')


for epoch in range(1, 20 + 1):
    train(epoch)
    test()

# Visualize the STN transformation on some input batch
visualize_stn()

plt.ioff()
plt.show()
```

Out:

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.334482
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.978787

Test set: Average loss: 0.2343, Accuracy: 9368/10000 (94%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.530352
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.569300

Test set: Average loss: 0.2207, Accuracy: 9317/10000 (93%)

Train Epoch: 3 [0/60000 (0%)]   Loss: 0.657171
Train Epoch: 3 [32000/60000 (53%)]      Loss: 0.293136

Test set: Average loss: 0.1030, Accuracy: 9714/10000 (97%)

Train Epoch: 4 [0/60000 (0%)]   Loss: 0.201117
Train Epoch: 4 [32000/60000 (53%)]      Loss: 0.263720

Test set: Average loss: 0.0811, Accuracy: 9754/10000 (98%)

Train Epoch: 5 [0/60000 (0%)]   Loss: 0.312747
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.172266

Test set: Average loss: 0.0653, Accuracy: 9815/10000 (98%)
```

```
Train Epoch: 6 [0/60000 (0%)]   Loss: 0.100500
Train Epoch: 6 [32000/60000 (53%)]      Loss: 0.368883

Test set: Average loss: 0.0660, Accuracy: 9810/10000 (98%)

Train Epoch: 7 [0/60000 (0%)]   Loss: 0.220224
Train Epoch: 7 [32000/60000 (53%)]      Loss: 0.186766

Test set: Average loss: 0.0565, Accuracy: 9850/10000 (98%)

Train Epoch: 8 [0/60000 (0%)]   Loss: 0.123097
Train Epoch: 8 [32000/60000 (53%)]      Loss: 0.127365

Test set: Average loss: 0.0616, Accuracy: 9815/10000 (98%)

Train Epoch: 9 [0/60000 (0%)]   Loss: 0.144827
Train Epoch: 9 [32000/60000 (53%)]      Loss: 0.262780

Test set: Average loss: 0.0490, Accuracy: 9857/10000 (99%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.117153
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.323490

Test set: Average loss: 0.0570, Accuracy: 9830/10000 (98%)

Train Epoch: 11 [0/60000 (0%)]  Loss: 0.187886
Train Epoch: 11 [32000/60000 (53%)]     Loss: 0.233467

Test set: Average loss: 0.0644, Accuracy: 9815/10000 (98%)

Train Epoch: 12 [0/60000 (0%)]  Loss: 0.180966
Train Epoch: 12 [32000/60000 (53%)]     Loss: 0.285183

Test set: Average loss: 0.2058, Accuracy: 9359/10000 (94%)

Train Epoch: 13 [0/60000 (0%)]  Loss: 0.741641
Train Epoch: 13 [32000/60000 (53%)]     Loss: 0.176156

Test set: Average loss: 0.0382, Accuracy: 9880/10000 (99%)

Train Epoch: 14 [0/60000 (0%)]  Loss: 0.108957
Train Epoch: 14 [32000/60000 (53%)]     Loss: 0.065435

Test set: Average loss: 0.0346, Accuracy: 9887/10000 (99%)

Train Epoch: 15 [0/60000 (0%)]  Loss: 0.203477
Train Epoch: 15 [32000/60000 (53%)]     Loss: 0.151954

Test set: Average loss: 0.0418, Accuracy: 9881/10000 (99%)

Train Epoch: 16 [0/60000 (0%)]  Loss: 0.037837
Train Epoch: 16 [32000/60000 (53%)]     Loss: 0.230621

Test set: Average loss: 0.0389, Accuracy: 9879/10000 (99%)

Train Epoch: 17 [0/60000 (0%)]  Loss: 0.081300
Train Epoch: 17 [32000/60000 (53%)]     Loss: 0.221337
```

```
Test set: Average loss: 0.0394, Accuracy: 9881/10000 (99%)

Train Epoch: 18 [0/60000 (0%)]  Loss: 0.027998
Train Epoch: 18 [32000/60000 (53%)]     Loss: 0.063587

Test set: Average loss: 0.1219, Accuracy: 9643/10000 (96%)

Train Epoch: 19 [0/60000 (0%)]  Loss: 0.175178
Train Epoch: 19 [32000/60000 (53%)]     Loss: 0.042914

Test set: Average loss: 0.0388, Accuracy: 9891/10000 (99%)

Train Epoch: 20 [0/60000 (0%)]  Loss: 0.061590
Train Epoch: 20 [32000/60000 (53%)]     Loss: 0.081851

Test set: Average loss: 0.0389, Accuracy: 9876/10000 (99%)
```

Total running time of the script: (18 minutes 11.545 seconds)

Download Python source code: spatial_transformer_tutorial.py

Download Jupyter notebook: spatial_transformer_tutorial.ipynb

Generated by Sphinx-Gallery

CHAPTER

THREE

ADVANCED TUTORIALS

3.1 Neural Transfer with PyTorch

Author: Alexis Jacq

3.1.1 Introduction

Welcome! This tutorial explains how to impletment the Neural-Style algorithm developed by Leon A. Gatys, Alexander S. Ecker and Matthias Bethge.

Neural what?

The Neural-Style, or Neural-Transfer, is an algorithm that takes as input a content-image (e.g. a tortle), a style-image (e.g. artistic waves) and return the content of the content-image as if it was 'painted' using the artistic style of the style-image:

How does it work?

The principle is simple: we define two distances, one for the content (D_C) and one for the style (D_S). D_C measures how different the content is between two images, while D_S measures how different the style is between two images. Then, we take a third image, the input, (e.g. a with noise), and we transform it in order to both minimize its content-distance with the content-image and its style-distance with the style-image.

OK. How does it work?

Well, going further requires some mathematics. Let C_{nn} be a pre-trained deep convolutional neural network and X be any image. $C_{nn}(X)$ is the network fed by X (containing feature maps at all layers). Let $F_{XL} \in C_{nn}(X)$ be the feature maps at depth layer L, all vectorized and concatenated in one single vector. We simply define the content of X

at layer L by F_{XL}. Then, if Y is another image of same the size than X, we define the distance of content at layer L as follow:

$$D_C^L(X,Y) = \|F_{XL} - F_{YL}\|^2 = \sum_i (F_{XL}(i) - F_{YL}(i))^2$$

Where $F_{XL}(i)$ is the i^{th} element of F_{XL}. The style is a bit less trivial to define. Let F_{XL}^k with $k \leq K$ be the vectorized k^{th} of the K feature maps at layer L. The style G_{XL} of X at layer L is defined by the Gram produce of all vectorized feature maps F_{XL}^k with $k \leq K$. In other words, G_{XL} is a KxK matrix and the element $G_{XL}(k,l)$ at the k^{th} line and l^{th} column of G_{XL} is the vectorial produce between F_{XL}^k and F_{XL}^l :

$$G_{XL}(k,l) = \langle F_{XL}^k, F_{XL}^l \rangle = \sum_i F_{XL}^k(i).F_{XL}^l(i)$$

Where $F_{XL}^k(i)$ is the i^{th} element of F_{XL}^k. We can see $G_{XL}(k,l)$ as a measure of the correlation between feature maps k and l. In that way, G_{XL} represents the correlation matrix of feature maps of X at layer L. Note that the size of G_{XL} only depends on the number of feature maps, not on the size of X. Then, if Y is another image *of any size*, we define the distance of style at layer L as follow:

$$D_S^L(X,Y) = \|G_{XL} - G_{YL}\|^2 = \sum_{k,l} (G_{XL}(k,l) - G_{YL}(k,l))^2$$

In order to minimize in one shot $D_C(X,C)$ between a variable image X and target content-image C and $D_S(X,S)$ between X and target style-image S, both computed at several layers , we compute and sum the gradients (derivative with respect to X) of each distance at each wanted layer:

$$\nabla_{extittotal}(X,S,C) = \sum_{L_C} w_{CL_C}.\nabla_{extitcontent}^{L_C}(X,C) + \sum_{L_S} w_{SL_S}.\nabla_{extitstyle}^{L_S}(X,S)$$

Where L_C and L_S are respectivement the wanted layers (arbitrary stated) of content and style and w_{CL_C} and w_{SL_S} the weights (arbitrary stated) associated with the style or the content at each wanted layer. Then, we run a gradient descent over X:

$$X \leftarrow X - \alpha \nabla_{extittotal}(X,S,C)$$

Ok. That's enough with maths. If you want to go deeper (how to compute the gradients) **we encourage you to read the original paper** by Leon A. Gatys and AL, where everything is much better and much clearer explained.

For our implementation in PyTorch, we already have everything we need: indeed, with PyTorch, all the gradients are automatically and dynamically computed for you (while you use functions from the library). This is why the implementation of this algorithm becomes very comfortable with PyTorch.

3.1.2 PyTorch implementation

If you are not sure to understand all the mathematics above, you will probably get it by implementing it. If you are discovering PyTorch, we recommend you to first read this *Introduction to PyTorch*.

Packages

We will have recourse to the following packages:

- `torch`, `torch.nn`, `numpy` (indispensables packages for neural networks with PyTorch)
- `torch.autograd.Variable` (dynamic computation of the gradient wrt a variable)
- `torch.optim` (efficient gradient descents)

- `PIL`, `PIL.Image`, `matplotlib.pyplot` (load and display images)
- `torchvision.transforms` (treat PIL images and transform into torch tensors)
- `torchvision.models` (train or load pre-trained models)
- `copy` (to deep copy the models; system package)

```
from __future__ import print_function

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

from PIL import Image
import matplotlib.pyplot as plt

import torchvision.transforms as transforms
import torchvision.models as models

import copy
```

Cuda

If you have a GPU on your computer, it is preferable to run the algorithm on it, especially if you want to try larger networks (like VGG). For this, we have `torch.cuda.is_available()` that returns `True` if you computer has an available GPU. Then, we can use method `.cuda()` that moves allocated proccesses associated with a module from the CPU to the GPU. When we want to move back this module to the CPU (e.g. to use numpy), we use the `.cpu()` method. Finally, `.type(dtype)` will be use to convert a `torch.FloatTensor` into `torch.cuda.FloatTensor` to feed GPU processes.

```
use_cuda = torch.cuda.is_available()
dtype = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
```

Load images

In order to simplify the implementation, let's start by importing a style and a content image of the same dimentions. We then scale them to the desired output image size (128 or 512 in the example, depending on gpu availablity) and transform them into torch tensors, ready to feed a neural network:

Note: Here are links to download the images required to run the tutorial: picasso.jpg and dancing.jpg. Download these two images and add them to a directory with name `images`

```
# desired size of the output image
imsize = 512 if use_cuda else 128  # use small size if no gpu

loader = transforms.Compose([
    transforms.Scale(imsize),  # scale imported image
    transforms.ToTensor()])  # transform it into a torch tensor


def image_loader(image_name):
    image = Image.open(image_name)
```

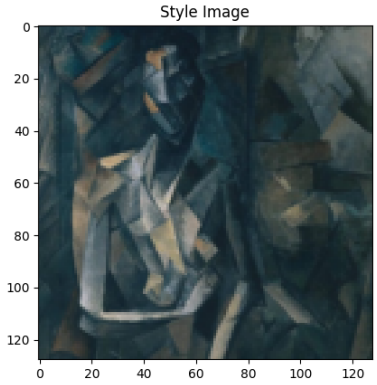
```
    image = Variable(loader(image))
    # fake batch dimension required to fit network's input dimensions
    image = image.unsqueeze(0)
    return image


style_img = image_loader("images/picasso.jpg").type(dtype)
content_img = image_loader("images/dancing.jpg").type(dtype)

assert style_img.size() == content_img.size(), \
    "we need to import style and content images of the same size"
```

Imported PIL images has values between 0 and 255. Transformed into torch tensors, their values are between 0 and 1. This is an important detail: neural networks from torch library are trained with 0-1 tensor image. If you try to feed the networks with 0-255 tensor images the activated feature maps will have no sense. This is not the case with pre-trained networks from the Caffe library: they are trained with 0-255 tensor images.

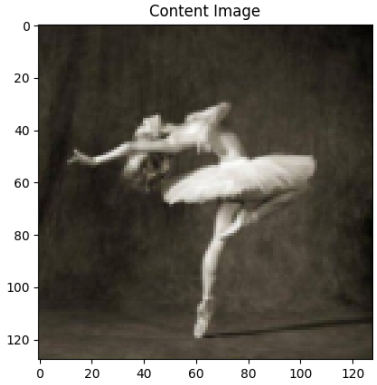
Display images

We will use `plt.imshow` to display images. So we need to first reconvert them into PIL images:

```
unloader = transforms.ToPILImage()  # reconvert into PIL image

plt.ion()

def imshow(tensor, title=None):
    image = tensor.clone().cpu()  # we clone the tensor to not do changes on it
    image = image.view(3, imsize, imsize)  # remove the fake batch dimension
    image = unloader(image)
    plt.imshow(image)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated


plt.figure()
imshow(style_img.data, title='Style Image')

plt.figure()
imshow(content_img.data, title='Content Image')
```



-



-

Content loss

The content loss is a function that takes as input the feature maps F_{XL} at a layer L in a network fed by X and return the weigthed content distance $w_{CL}.D_C^L(X,C)$ between this image and the content image. Hence, the weight w_{CL} and the target content F_{CL} are parameters of the function. We implement this function as a torch module with a constructor that takes these parameters as input. The distance $\|F_{XL} - F_{YL}\|^2$ is the Mean Square Error between the two sets of feature maps, that can be computed using a criterion `nn.MSELoss` stated as a third parameter.

We will add our content losses at each desired layer as additive modules of the neural network. That way, each time we will feed the network with an input image X, all the content losses will be computed at the desired layers and, thanks to autograd, all the gradients will be computed. For that, we just need to make the `forward` method of our module returning the input: the module becomes a ''transparent layer" of the neural network. The computed loss is saved as a parameter of the module.

Finally, we define a fake `backward` method, that just call the backward method of `nn.MSELoss` in order to reconstruct the gradient. This method returns the computed loss: this will be useful when running the gradient descent in order to display the evolution of style and content losses.

```
class ContentLoss(nn.Module):

    def __init__(self, target, weight):
        super(ContentLoss, self).__init__()
        # we 'detach' the target content from the tree used
        self.target = target.detach() * weight
        # to dynamically compute the gradient: this is a stated value,
        # not a variable. Otherwise the forward method of the criterion
        # will throw an error.
        self.weight = weight
        self.criterion = nn.MSELoss()

    def forward(self, input):
        self.loss = self.criterion(input * self.weight, self.target)
        self.output = input
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

Note: **Important detail**: this module, although it is named `ContentLoss`, is not a true PyTorch Loss function.

If you want to define your content loss as a PyTorch Loss, you have to create a PyTorch autograd Function and to recompute/implement the gradient by the hand in the `backward` method.

Style loss

For the style loss, we need first to define a module that compute the gram produce G_{XL} given the feature maps F_{XL} of the neural network fed by X, at layer L. Let $\hat{F}_{XL}$ be the re-shaped version of F_{XL} into a KxN matrix, where K is the number of feature maps at layer L and N the lenght of any vectorized feature map F^k_{XL}. The k^{th} line of $\hat{F}_{XL}$ is F^k_{XL}. We let you check that $\hat{F}_{XL} \cdot \hat{F}^T_{XL} = G_{XL}$. Given that, it becomes easy to implement our module:

```
class GramMatrix(nn.Module):

    def forward(self, input):
        a, b, c, d = input.size()  # a=batch size(=1)
        # b=number of feature maps
        # (c,d)=dimensions of a f. map (N=c*d)

        features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL

        G = torch.mm(features, features.t())  # compute the gram product

        # we 'normalize' the values of the gram matrix
        # by dividing by the number of element in each feature maps.
        return G.div(a * b * c * d)
```

The longer is the feature maps dimension N, the bigger are the values of the gram matrix. Therefore, if we don't normalize by N, the loss computed at the first layers (before pooling layers) will have much more importance during the gradient descent. We dont want that, since the most interesting style features are in the deepest layers!

Then, the style loss module is implemented exactly the same way than the content loss module, but we have to add the `gramMatrix` as a parameter:

```
class StyleLoss(nn.Module):

    def __init__(self, target, weight):
        super(StyleLoss, self).__init__()
        self.target = target.detach() * weight
        self.weight = weight
        self.gram = GramMatrix()
        self.criterion = nn.MSELoss()

    def forward(self, input):
        self.output = input.clone()
        self.G = self.gram(input)
        self.G.mul_(self.weight)
        self.loss = self.criterion(self.G, self.target)
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss
```

Load the neural network

Now, we have to import a pre-trained neural network. As in the paper, we are going to use a pretrained VGG network with 19 layers (VGG19).

PyTorch's implementation of VGG is a module divided in two child `Sequential` modules: `features` (containing convolution and pooling layers) and `classifier` (containing fully connected layers). We are just interested by `features`:

```
cnn = models.vgg19(pretrained=True).features

# move it to the GPU if possible:
if use_cuda:
    cnn = cnn.cuda()
```

A `Sequential` module contains an ordered list of child modules. For instance, `vgg19.features` contains a sequence (Conv2d, ReLU, Maxpool2d, Conv2d, ReLU...) aligned in the right order of depth. As we said in *Content loss* section, we wand to add our style and content loss modules as additive 'transparent' layers in our network, at desired depths. For that, we construct a new `Sequential` module, in wich we are going to add modules from `vgg19` and our loss modules in the right order:

```
# desired depth layers to compute style/content losses :
content_layers_default = ['conv_4']
style_layers_default = ['conv_1', 'conv_2', 'conv_3', 'conv_4', 'conv_5']


def get_style_model_and_losses(cnn, style_img, content_img,
                               style_weight=1000, content_weight=1,
                               content_layers=content_layers_default,
                               style_layers=style_layers_default):
    cnn = copy.deepcopy(cnn)

    # just in order to have an iterable access to or list of content/syle
    # losses
    content_losses = []
    style_losses = []

    model = nn.Sequential()  # the new Sequential module network
    gram = GramMatrix()  # we need a gram module in order to compute style targets

    # move these modules to the GPU if possible:
    if use_cuda:
        model = model.cuda()
        gram = gram.cuda()

    i = 1
    for layer in list(cnn):
        if isinstance(layer, nn.Conv2d):
            name = "conv_" + str(i)
            model.add_module(name, layer)

            if name in content_layers:
                # add content loss:
                target = model(content_img).clone()
                content_loss = ContentLoss(target, content_weight)
                model.add_module("content_loss_" + str(i), content_loss)
                content_losses.append(content_loss)
```

```
            if name in style_layers:
                # add style loss:
                target_feature = model(style_img).clone()
                target_feature_gram = gram(target_feature)
                style_loss = StyleLoss(target_feature_gram, style_weight)
                model.add_module("style_loss_" + str(i), style_loss)
                style_losses.append(style_loss)

        if isinstance(layer, nn.ReLU):
            name = "relu_" + str(i)
            model.add_module(name, layer)

            if name in content_layers:
                # add content loss:
                target = model(content_img).clone()
                content_loss = ContentLoss(target, content_weight)
                model.add_module("content_loss_" + str(i), content_loss)
                content_losses.append(content_loss)

            if name in style_layers:
                # add style loss:
                target_feature = model(style_img).clone()
                target_feature_gram = gram(target_feature)
                style_loss = StyleLoss(target_feature_gram, style_weight)
                model.add_module("style_loss_" + str(i), style_loss)
                style_losses.append(style_loss)

            i += 1

        if isinstance(layer, nn.MaxPool2d):
            name = "pool_" + str(i)
            model.add_module(name, layer)  # ***

    return model, style_losses, content_losses
```

Note: In the paper they recommend to change max pooling layers into average pooling. With AlexNet, that is a small network compared to VGG19 used in the paper, we are not going to see any difference of quality in the result. However, you can use these lines instead if you want to do this substitution:

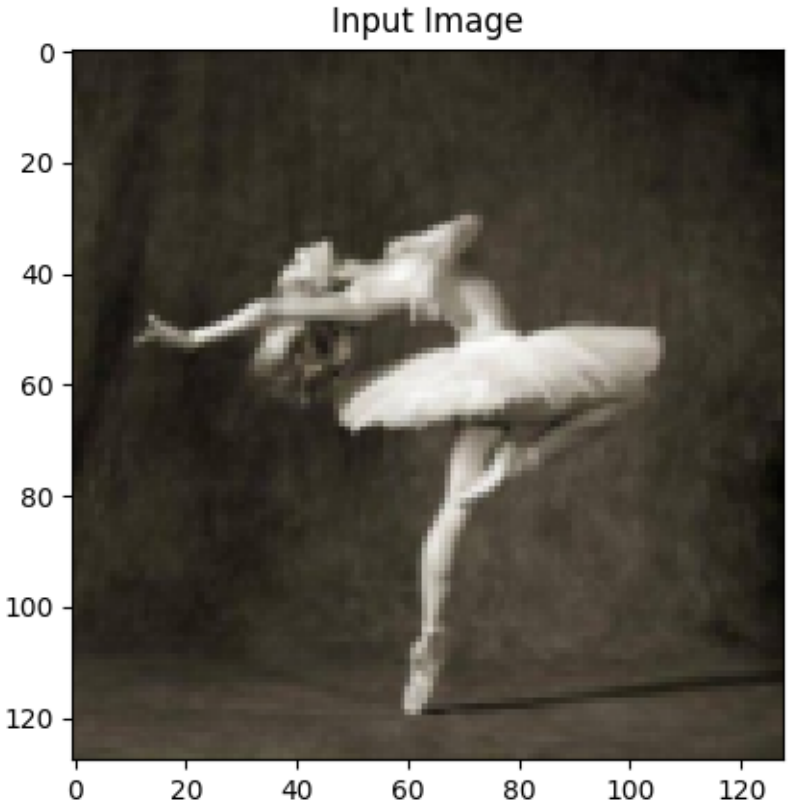
```
# avgpool = nn.AvgPool2d(kernel_size=layer.kernel_size,
#                        stride=layer.stride, padding = layer.padding)
# model.add_module(name,avgpool)
```

Input image

Again, in order to simplify the code, we take an image of the same dimensions than content and style images. This image can be a white noise, or it can also be a copy of the content-image.

```
input_img = content_img.clone()
# if you want to use a white noise instead uncomment the below line:
# input_img = Variable(torch.randn(content_img.data.size())).type(dtype)

# add the original input image to the figure:
```

```
plt.figure()
imshow(input_img.data, title='Input Image')
```



Gradient descent

As Leon Gatys, the author of the algorithm, suggested here, we will use L-BFGS algorithm to run our gradient descent. Unlike training a network, we want to train the input image in order to minimise the content/style losses. We would like to simply create a PyTorch L-BFGS optimizer, passing our image as the variable to optimize. But `optim.LBFGS` takes as first argument a list of PyTorch `Variable` that require gradient. Our input image is a `Variable` but is not a leaf of the tree that requires computation of gradients. In order to show that this variable requires a gradient, a possibility is to construct a `Parameter` object from the input image. Then, we just give a list containing this `Parameter` to the optimizer's constructor:

```
def get_input_param_optimizer(input_img):
    # this line to show that input is a parameter that requires a gradient
    input_param = nn.Parameter(input_img.data)
    optimizer = optim.LBFGS([input_param])
    return input_param, optimizer
```

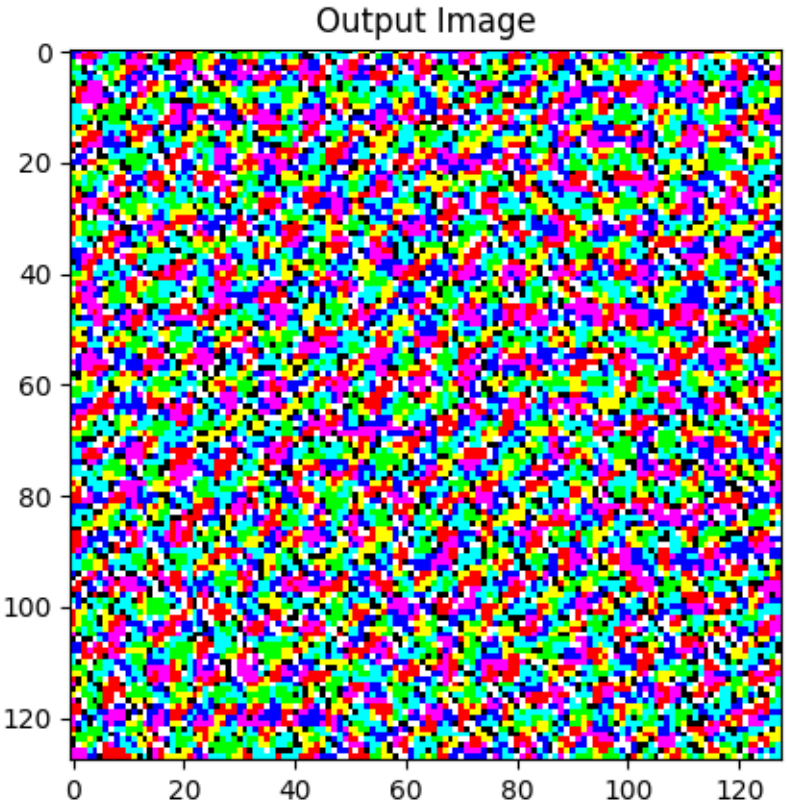
Last step: the loop of gradient descent. At each step, we must feed the network with the updated input in order to compute the new losses, we must run the `backward` methods of each loss to dynamically compute their gradients and perform the step of gradient descent. The optimizer requires as argument a "closure": a function that reevaluates the model and returns the loss.

However, there's a small catch. The optimized image may take its values between $-\infty$ and $+\infty$ instead of staying between 0 and 1. In other words, the image might be well optimized and have absurd values. In fact, we must perform an optimization under constraints in order to keep having right vaues into our input image. There is a simple solution: at each step, to correct the image to maintain its values into the 0-1 interval.

```
def run_style_transfer(cnn, content_img, style_img, input_img, num_steps=300,
                       style_weight=1000, content_weight=1):
    """Run the style transfer."""
    print('Building the style transfer model..')
    model, style_losses, content_losses = get_style_model_and_losses(cnn,
        style_img, content_img, style_weight, content_weight)
    input_param, optimizer = get_input_param_optimizer(input_img)

    print('Optimizing..')
    run = [0]
    while run[0] <= num_steps:

        def closure():
            # correct the values of updated input image
            input_param.data.clamp_(0, 1)

            optimizer.zero_grad()
            model(input_param)
            style_score = 0
            content_score = 0

            for sl in style_losses:
                style_score += sl.backward()
            for cl in content_losses:
                content_score += cl.backward()

            run[0] += 1
            if run[0] % 50 == 0:
                print("run {}:".format(run))
                print('Style Loss : {:4f} Content Loss: {:4f}'.format(
                    style_score.data[0], content_score.data[0]))
                print()

            return style_score + content_score

        optimizer.step(closure)

    # a last correction...
    input_param.data.clamp_(0, 1)

    return input_param.data
```

Finally, run the algorithm

```
output = run_style_transfer(cnn, content_img, style_img, input_img)

plt.figure()
imshow(output, title='Output Image')

# sphinx_gallery_thumbnail_number = 4
plt.ioff()
plt.show()
```



Out:

```
Building the style transfer model..
Optimizing..
run [50]:
Style Loss : 1046161.437500 Content Loss: 71.575310

run [100]:
Style Loss : 1046161.437500 Content Loss: 71.575310

run [150]:
Style Loss : 1046161.437500 Content Loss: 71.575310

run [200]:
Style Loss : 1046161.437500 Content Loss: 71.575310

run [250]:
Style Loss : 1046161.437500 Content Loss: 71.575310

run [300]:
Style Loss : 1046161.437500 Content Loss: 71.575310
```

Total running time of the script: (5 minutes 29.058 seconds)

3.2 Creating extensions using numpy and scipy

Author: Adam Paszke

In this tutorial, we shall go through two tasks:

1. Create a neural network layer with no parameters.
    - This calls into **numpy** as part of it's implementation
2. Create a neural network layer that has learnable weights
    - This calls into **SciPy** as part of it's implementation

```
import torch
from torch.autograd import Function
from torch.autograd import Variable
```

3.2.1 Parameter-less example

This layer doesn't particularly do anything useful or mathematically correct.

It is aptly named BadFFTFunction

Layer Implementation

```
from numpy.fft import rfft2, irfft2


class BadFFTFunction(Function):

    def forward(self, input):
        numpy_input = input.numpy()
        result = abs(rfft2(numpy_input))
        return torch.FloatTensor(result)

    def backward(self, grad_output):
        numpy_go = grad_output.numpy()
        result = irfft2(numpy_go)
        return torch.FloatTensor(result)

# since this layer does not have any parameters, we can
# simply declare this as a function, rather than as an nn.Module class


def incorrect_fft(input):
    return BadFFTFunction()(input)
```

Example usage of the created layer:

```
input = Variable(torch.randn(8, 8), requires_grad=True)
result = incorrect_fft(input)
print(result.data)
result.backward(torch.randn(result.size()))
print(input.grad)
```

Out:

```
1.1785   5.5621  10.4863   6.1444   2.2919
  2.5107  10.0722   4.4866   5.0767   1.5958
 10.0418   7.7827   7.9334   2.7846   8.0642
  5.3870   6.7392   5.1057   1.1831   9.8985
  4.1078  13.1084   3.9163   8.1475  17.7848
  5.3870   2.2938   6.6977   2.4565   9.8985
 10.0418   7.3248  11.5897   9.4735   8.0642
  2.5107  10.1200   6.4771  12.7477   1.5958
[torch.FloatTensor of size 8x5]

Variable containing:
 0.2227 -0.0561  0.0225  0.1155 -0.3444  0.1155  0.0225 -0.0561
 0.1183 -0.1768 -0.1348  0.0205 -0.0008  0.0390  0.1087 -0.1144
 0.0031 -0.1295  0.1566  0.0311 -0.1683 -0.0525 -0.0473  0.0824
-0.2096  0.1023  0.1965 -0.1857  0.1088  0.0931 -0.1085 -0.1349
 0.0868  0.1321 -0.0937 -0.1482 -0.1189 -0.1482 -0.0937  0.1321
-0.2096 -0.1349 -0.1085  0.0931  0.1088 -0.1857  0.1965  0.1023
 0.0031  0.0824 -0.0473 -0.0525 -0.1683  0.0311  0.1566 -0.1295
 0.1183 -0.1144  0.1087  0.0390 -0.0008  0.0205 -0.1348 -0.1768
[torch.FloatTensor of size 8x8]
```

3.2.2 Parametrized example

This implements a layer with learnable weights.

It implements the Cross-correlation with a learnable kernel.

In deep learning literature, it's confusingly referred to as Convolution.

The backward computes the gradients wrt the input and gradients wrt the filter.

Implementation:

Please Note that the implementation serves as an illustration, and we did not verify it's correctness

```
from scipy.signal import convolve2d, correlate2d
from torch.nn.modules.module import Module
from torch.nn.parameter import Parameter


class ScipyConv2dFunction(Function):
    @staticmethod
    def forward(ctx, input, filter):
        result = correlate2d(input.numpy(), filter.numpy(), mode='valid')
        ctx.save_for_backward(input, filter)
        return torch.FloatTensor(result)

    @staticmethod
    def backward(ctx, grad_output):
        input, filter = ctx.saved_tensors
        grad_output = grad_output.data
        grad_input = convolve2d(grad_output.numpy(), filter.t().numpy(), mode='full')
        grad_filter = convolve2d(input.numpy(), grad_output.numpy(), mode='valid')

        return Variable(torch.FloatTensor(grad_input)), \
            Variable(torch.FloatTensor(grad_filter))
```

```
class ScipyConv2d(Module):

    def __init__(self, kh, kw):
        super(ScipyConv2d, self).__init__()
        self.filter = Parameter(torch.randn(kh, kw))

    def forward(self, input):
        return ScipyConv2dFunction.apply(input, self.filter)
```

Example usage:

```
module = ScipyConv2d(3, 3)
print(list(module.parameters()))
input = Variable(torch.randn(10, 10), requires_grad=True)
output = module(input)
print(output)
output.backward(torch.randn(8, 8))
print(input.grad)
```

Out:

```
[Parameter containing:
-0.5763 -0.3928  0.5131
-0.2001 -0.5723 -0.5581
 0.7312  0.2859 -0.9594
[torch.FloatTensor of size 3x3]
]
Variable containing:
 1.1538 -1.6958  2.0817  2.0357  1.6593 -0.3197 -1.2532  1.9277
 0.7446  1.3001  0.7509 -0.4676 -0.0675 -0.7282  1.0574 -0.9021
-0.0073  1.0119  2.0335  2.8383 -2.1522  3.4802  4.2341 -1.8476
-0.9056  0.1508 -1.0507  2.2670  0.0192  1.6211  2.0648  1.6976
 0.2703  0.8417  0.0914  0.1321 -0.6875 -0.2092  0.4475  0.5156
 0.0118  0.2128  1.6066  1.0431  0.2998 -1.7713 -0.9442 -1.7070
 1.0011 -1.2979  0.1056  3.5285  0.1001 -1.8432  1.2209 -2.2866
 2.3896 -3.1380 -4.5054  3.4745  0.8358 -0.4329  0.7343 -0.0214
[torch.FloatTensor of size 8x8]

Variable containing:
-0.1651  0.3374  0.7349  0.2396 -0.0082 -0.4381  0.5537  0.2036 -0.9651 -0.2751
-0.3122  1.1737  0.8827 -0.5557  1.5018 -0.0805  1.4186  0.3373 -1.3062  0.3426
 0.1414 -0.2034  1.3092  0.5141  0.9205  1.5618 -0.4865  0.5371  2.0304  0.6145
-0.0707 -1.6120  1.8295  2.2535 -0.5228  1.4510 -0.2148  0.8304  0.6448 -0.9467
 0.6262  0.1940 -1.9504  0.6356 -0.2805  0.6264  3.6983  0.0244 -0.9131 -1.0811
 0.4057  0.9508  0.8995  0.1696 -2.0125 -0.8552  0.2905 -0.6813  1.6649  1.4420
-1.3284  1.4344  3.9961  0.5244  0.4238 -1.0757 -0.6263 -0.5733 -1.6633  1.8329
-0.8189 -0.2661  0.8392  1.1481  2.3459  0.1140 -0.4206  1.9613  0.3358 -1.6390
-0.7834 -1.5779  0.5551  0.9997  2.3109 -0.4114 -2.2060  0.3171  0.9720 -0.3667
 1.1200 -1.7671 -0.8483  0.3379 -1.6428  1.1887  0.8550 -2.5291 -1.6808  0.1133
[torch.FloatTensor of size 10x10]
```

Total running time of the script: (0 minutes 0.006 seconds)

Download Python source code: numpy_extensions_tutorial.py

Download Jupyter notebook: numpy_extensions_tutorial.ipynb

Generated by Sphinx-Gallery

3.3 Transfering a model from PyTorch to Caffe2 and Mobile using ONNX

In this tutorial, we describe how to use ONNX to convert a model defined in PyTorch into the ONNX format and then load it into Caffe2. Once in Caffe2, we can run the model to double-check it was exported correctly, and we then show how to use Caffe2 features such as mobile exporter for executing the model on mobile devices.

For this tutorial, you will need to install onnx, onnx-caffe2 and Caffe2. You can get binary builds of onnx and onnx-caffe2 with `conda install -c ezyang onnx onnx-caffe2`.

NOTE: This tutorial needs PyTorch master branch which can be installed by following the instructions here

```
# Some standard imports
import io
import numpy as np

from torch import nn
from torch.autograd import Variable
import torch.utils.model_zoo as model_zoo
import torch.onnx
```

Super-resolution is a way of increasing the resolution of images, videos and is widely used in image processing or video editing. For this tutorial, we will first use a small super-resolution model with a dummy input.

First, let's create a SuperResolution model in PyTorch. This model comes directly from PyTorch's examples without modification:

```
# Super Resolution model definition in PyTorch
import torch.nn as nn
import torch.nn.init as init


class SuperResolutionNet(nn.Module):
    def __init__(self, upscale_factor, inplace=False):
        super(SuperResolutionNet, self).__init__()

        self.relu = nn.ReLU(inplace=inplace)
        self.conv1 = nn.Conv2d(1, 64, (5, 5), (1, 1), (2, 2))
        self.conv2 = nn.Conv2d(64, 64, (3, 3), (1, 1), (1, 1))
        self.conv3 = nn.Conv2d(64, 32, (3, 3), (1, 1), (1, 1))
        self.conv4 = nn.Conv2d(32, upscale_factor ** 2, (3, 3), (1, 1), (1, 1))
        self.pixel_shuffle = nn.PixelShuffle(upscale_factor)

        self._initialize_weights()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pixel_shuffle(self.conv4(x))
        return x

    def _initialize_weights(self):
        init.orthogonal(self.conv1.weight, init.calculate_gain('relu'))
        init.orthogonal(self.conv2.weight, init.calculate_gain('relu'))
        init.orthogonal(self.conv3.weight, init.calculate_gain('relu'))
        init.orthogonal(self.conv4.weight)
```

```
# Create the super-resolution model by using the above model definition.
torch_model = SuperResolutionNet(upscale_factor=3)
```

Ordinarily, you would now train this model; however, for this tutorial, we will instead download some pre-trained weights. Note that this model was not trained fully for good accuracy and is used here for demonstration purposes only.

```
# Load pretrained model weights
model_url = 'https://s3.amazonaws.com/pytorch/test_data/export/superres_epoch100-
↪44c6958e.pth'
batch_size = 1    # just a random number

# Initialize model with the pretrained weights
map_location = lambda storage, loc: storage
if torch.cuda.is_available():
    map_location = None
torch_model.load_state_dict(model_zoo.load_url(model_url, map_location=map_location))

# set the train mode to false since we will only run the forward pass.
torch_model.train(False)
```

Exporting a model in PyTorch works via tracing. To export a model, you call the `torch.onnx._export()` function. This will execute the model, recording a trace of what operators are used to compute the outputs. Because `_export` runs the model, we need provide an input tensor `x`. The values in this tensor are not important; it can be an image or a random tensor as long as it is the right size.

To learn more details about PyTorch's export interface, check out the torch.onnx documentation.

```
# Input to the model
x = Variable(torch.randn(batch_size, 1, 224, 224), requires_grad=True)

# Export the model
torch_out = torch.onnx._export(torch_model,             # model being run
                               x,                       # model input (or a tuple for␣
↪multiple inputs)
                               "super_resolution.onnx", # where to save the model␣
↪(can be a file or file-like object)
                               export_params=True)      # store the trained parameter␣
↪weights inside the model file
```

`torch_out` is the output after executing the model. Normally you can ignore this output, but here we will use it to verify that the model we exported computes the same values when run in Caffe2.

Now let's take the ONNX representation and use it in Caffe2. This part can normally be done in a separate process or on another machine, but we will continue in the same process so that we can verify that Caffe2 and PyTorch are computing the same value for the network:

```
import onnx
import onnx_caffe2.backend

# Load the ONNX ModelProto object. model is a standard Python protobuf object
model = onnx.load("super_resolution.onnx")

# prepare the caffe2 backend for executing the model this converts the ONNX model␣
↪into a
# Caffe2 NetDef that can execute it. Other ONNX backends, like one for CNTK will be
# availiable soon.
prepared_backend = onnx_caffe2.backend.prepare(model)
```

```
# run the model in Caffe2

# Construct a map from input names to Tensor data.
# The graph of the model itself contains inputs for all weight parameters, after the
↪input image.
# Since the weights are already embedded, we just need to pass the input image.
# Set the first input.
W = {model.graph.input[0].name: x.data.numpy()}

# Run the Caffe2 net:
c2_out = prepared_backend.run(W)[0]

# Verify the numerical correctness upto 3 decimal places
np.testing.assert_almost_equal(torch_out.data.cpu().numpy(), c2_out, decimal=3)

print("Exported model has been executed on Caffe2 backend, and the result looks good!
↪")
```

We should see that the output of PyTorch and Caffe2 runs match numerically up to 3 decimal places. As a side-note, if they do not match then there is an issue that the operators in Caffe2 and PyTorch are implemented differently and please contact us in that case.

3.3.1 Transfering SRResNet using ONNX

Using the same process as above, we also transferred an interesting new model "SRResNet" for super-resolution presented in this paper (thanks to the authors at Twitter for providing us code and pretrained parameters for the purpose of this tutorial). The model definition and a pre-trained model can be found here. Below is what SRResNet model input, output looks like.

Original low-resolution image

The same original image zoomed 4 times

Output of SRResNet model

3.3.2 Running the model on mobile devices

So far we have exported a model from PyTorch and shown how to load it and run it in Caffe2. Now that the model is loaded in Caffe2, we can convert it into a format suitable for running on mobile devices.

We will use Caffe2's mobile_exporter to generate the two model protobufs that can run on mobile. The first is used to initialize the network with the correct weights, and the second actual runs executes the model. We will continue to use the small super-resolution model for the rest of this tutorial.

```
# extract the workspace and the model proto from the internal representation
c2_workspace = prepared_backend.workspace
c2_model = prepared_backend.predict_net

# Now import the caffe2 mobile exporter
from caffe2.python.predictor import mobile_exporter

# call the Export to get the predict_net, init_net. These nets are needed for running␣
↪things on mobile
init_net, predict_net = mobile_exporter.Export(c2_workspace, c2_model, c2_model.
↪external_input)

# Let's also save the init_net and predict_net to a file that we will later use for␣
↪running them on mobile
with open('init_net.pb', "wb") as fopen:
    fopen.write(init_net.SerializeToString())
with open('predict_net.pb', "wb") as fopen:
    fopen.write(predict_net.SerializeToString())
```

`init_net` has the model parameters and the model input embedded in it and `predict_net` will be used to guide the `init_net` execution at run-time. In this tutorial, we will use the `init_net` and `predict_net` generated above and run them in both normal Caffe2 backend and mobile and verify that the output high-resolution cat image produced in both runs is the same.

For this tutorial, we will use a famous cat image used widely which looks like below

```
# Some standard imports
from caffe2.proto import caffe2_pb2
from caffe2.python import core, net_drawer, net_printer, visualize, workspace, utils

import numpy as np
import os
import subprocess
from PIL import Image
from matplotlib import pyplot
from skimage import io, transform
```

First, let's load the image, pre-process it using standard skimage python library. Note that this preprocessing is the standard practice of processing data for training/testing neural networks.

```
# load the image
img_in = io.imread("./_static/img/cat.jpg")

# resize the image to dimensions 224x224
img = transform.resize(img_in, [224, 224])

# save this resized image to be used as input to the model
io.imsave("./_static/img/cat_224x224.jpg", img)
```

Now, as a next step, let's take the resized cat image and run the super-resolution model in Caffe2 backend and save the output image. The image processing steps below have been adopted from PyTorch implementation of super-resolution model here

```
# load the resized image and convert it to Ybr format
img = Image.open("./_static/img/cat_224x224.jpg")
img_ycbcr = img.convert('YCbCr')
img_y, img_cb, img_cr = img_ycbcr.split()

# Let's run the mobile nets that we generated above so that caffe2 workspace is␣
↪properly initialized
workspace.RunNetOnce(init_net)
workspace.RunNetOnce(predict_net)

# Caffe2 has a nice net_printer to be able to inspect what the net looks like and␣
↪identify
# what our input and output blob names are.
print(net_printer.to_string(predict_net))
```

From the above output, we can see that input is named "9" and output is named "27"(it is a little bit weird that we will have numbers as blob names but this is because the tracing JIT produces numbered entries for the models)

```
# Now, let's also pass in the resized cat image for processing by the model.
workspace.FeedBlob("9", np.array(img_y)[np.newaxis, np.newaxis, :, :].astype(np.
↪float32))

# run the predict_net to get the model output
workspace.RunNetOnce(predict_net)

# Now let's get the model output blob
img_out = workspace.FetchBlob("27")
```

Now, we'll refer back to the post-processing steps in PyTorch implementation of super-resolution model here to construct back the final output image and save the image.

```
img_out_y = Image.fromarray(np.uint8((img_out[0, 0]).clip(0, 255)), mode='L')

# get the output image follow post-processing step from PyTorch implementation
final_img = Image.merge(
    "YCbCr", [
        img_out_y,
        img_cb.resize(img_out_y.size, Image.BICUBIC),
        img_cr.resize(img_out_y.size, Image.BICUBIC),
    ]).convert("RGB")

# Save the image, we will compare this with the output image from mobile device
```

```
final_img.save("./_static/img/cat_superres.jpg")
```

We have finished running our mobile nets in pure Caffe2 backend and now, let's execute the model on an Android device and get the model output.

NOTE: for Android development, adb shell is needed otherwise the following section of tutorial will not run.

In our first step of runnig model on mobile, we will push a native speed benchmark binary for mobile device to adb. This binary can execute the model on mobile and also export the model output that we can retrieve later. The binary is available here. In order to build the binary, execute the build_android.sh script following the instructions here.

NOTE: You need to have ANDROID_NDK installed and set your env variable ANDROID_NDK=path to ndk root

```
# let's first push a bunch of stuff to adb, specify the path for the binary
CAFFE2_MOBILE_BINARY = ('caffe2/binaries/speed_benchmark')

# we had saved our init_net and proto_net in steps above, we use them now.
# Push the binary and the model protos
os.system('adb push ' + CAFFE2_MOBILE_BINARY + ' /data/local/tmp/')
os.system('adb push init_net.pb /data/local/tmp')
os.system('adb push predict_net.pb /data/local/tmp')

# Let's serialize the input image blob to a blob proto and then send it to mobile for␣
↪execution.
with open("input.blobproto", "wb") as fid:
    fid.write(workspace.SerializeBlob("9"))

# push the input image blob to adb
os.system('adb push input.blobproto /data/local/tmp/')

# Now we run the net on mobile, look at the speed_benchmark --help for what various␣
↪options mean
os.system(
    'adb shell /data/local/tmp/speed_benchmark '                    # binary to␣
↪execute
    '--init_net=/data/local/tmp/super_resolution_mobile_init.pb '    # mobile init_net
    '--net=/data/local/tmp/super_resolution_mobile_predict.pb '      # mobile predict_
↪net
    '--input=9 '                                                    # name of our␣
↪input image blob
    '--input_file=/data/local/tmp/input.blobproto '                 # serialized␣
↪input image
    '--output_folder=/data/local/tmp '                              # destination␣
↪folder for saving mobile output
    '--output=27,9 '                                                # output blobs␣
↪we are interested in
    '--iter=1 '                                                     # number of net␣
↪iterations to execute
    '--caffe2_log_level=0 '
)

# get the model output from adb and save to a file
os.system('adb pull /data/local/tmp/27 ./output.blobproto')


# We can recover the output content and post-process the model using same steps as we␣
↪followed earlier
blob_proto = caffe2_pb2.BlobProto()
blob_proto.ParseFromString(open('./output.blobproto').read())
```

```
img_out = utils.Caffe2TensorToNumpyArray(blob_proto.tensor)
img_out_y = Image.fromarray(np.uint8((img_out[0,0]).clip(0, 255)), mode='L')
final_img = Image.merge(
    "YCbCr", [
        img_out_y,
        img_cb.resize(img_out_y.size, Image.BICUBIC),
        img_cr.resize(img_out_y.size, Image.BICUBIC),
    ]).convert("RGB")
final_img.save("./_static/img/cat_superres_mobile.jpg")
```

Now, you can compare the image `cat_superres.jpg` (model output from pure caffe2 backend execution) and `cat_superres_mobile.jpg` (model output from mobile execution) and see that both the images look same. If they don't look same, something went wrong with execution on mobile and in that case, please contact Caffe2 community. You should expect to see the output image to look like following:

Using the above steps, you can deploy your models on mobile easily. Also, for more information on caffe2 mobile backend, checkout caffe2-android-demo.

Total running time of the script: (0 minutes 0.000 seconds)

3.4 Custom C extensions for pytorch

Author: Soumith Chintala

3.4.1 Step 1. prepare your C code

First, you have to write your C functions.

Below you can find an example implementation of forward and backward functions of a module that adds its both inputs.

In your `.c` files you can include TH using an `#include <TH/TH.h>` directive, and THC using `#include <THC/THC.h>`.

ffi utils will make sure a compiler can find them during the build.

```
/* src/my_lib.c */
#include <TH/TH.h>

int my_lib_add_forward(THFloatTensor *input1, THFloatTensor *input2,
THFloatTensor *output)
{
    if (!THFloatTensor_isSameSizeAs(input1, input2))
        return 0;
    THFloatTensor_resizeAs(output, input1);
    THFloatTensor_cadd(output, input1, 1.0, input2);
    return 1;
}

int my_lib_add_backward(THFloatTensor *grad_output, THFloatTensor *grad_input)
{
    THFloatTensor_resizeAs(grad_input, grad_output);
    THFloatTensor_fill(grad_input, 1);
    return 1;
}
```

There are no constraints on the code, except that you will have to prepare a single header, which will list all functions want to call from python.

It will be used by the ffi utils to generate appropriate wrappers.

```
/* src/my_lib.h */
int my_lib_add_forward(THFloatTensor *input1, THFloatTensor *input2, THFloatTensor␣
↪*output);
int my_lib_add_backward(THFloatTensor *grad_output, THFloatTensor *grad_input);
```

Now, you'll need a super short file, that will build your custom extension:

```
# build.py
from torch.utils.ffi import create_extension
ffi = create_extension(
name='_ext.my_lib',
headers='src/my_lib.h',
sources=['src/my_lib.c'],
with_cuda=False
)
ffi.build()
```

3.4.2 Step 2: Include it in your Python code

After you run it, pytorch will create an `_ext` directory and put `my_lib` inside.

Package name can have an arbitrary number of packages preceding the final module name (including none). If the build succeeded you can import your extension just like a regular python file.

```
# functions/add.py
import torch
from torch.autograd import Function
from _ext import my_lib


class MyAddFunction(Function):
    def forward(self, input1, input2):
```

```
        output = torch.FloatTensor()
        my_lib.my_lib_add_forward(input1, input2, output)
        return output

    def backward(self, grad_output):
        grad_input = torch.FloatTensor()
        my_lib.my_lib_add_backward(grad_output, grad_input)
        return grad_input
```

```
# modules/add.py
from torch.nn import Module
from functions.add import MyAddFunction

class MyAddModule(Module):
    def forward(self, input1, input2):
        return MyAddFunction()(input1, input2)
```

```
# main.py
import torch
import torch.nn as nn
from torch.autograd import Variable
from modules.add import MyAddModule

class MyNetwork(nn.Module):
    def __init__(self):
        super(MyNetwork, self).__init__()
        self.add = MyAddModule()

    def forward(self, input1, input2):
        return self.add(input1, input2)

model = MyNetwork()
input1, input2 = Variable(torch.randn(5, 5)), Variable(torch.randn(5, 5))
print(model(input1, input2))
print(input1 + input2)
```