In [1]:

```
%matplotlib inline
```

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.

In [2]:

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

Construct a 5x3 matrix, uninitialized:

In [3]:

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

```
tensor([[ 2.2696e+09,  4.5807e-41,  2.2696e+09],
        [ 4.5807e-41,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00],
        [ 0.0000e+00,  0.0000e+00,  0.0000e+00]])
```

Construct a randomly initialized matrix:

In [4]:

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

```
tensor([[ 0.5462,  0.7124,  0.4470],
        [ 0.7190,  0.3102,  0.8786],
        [ 0.2969,  0.9215,  0.3329],
        [ 0.7350,  0.6148,  0.5322],
        [ 0.5491,  0.4307,  0.7958]])
```

Construct a matrix filled zeros and of dtype long:

In [5]:

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

```
tensor([[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]])
```

Construct a tensor directly from data:

In [6]:

```
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([ 5.5000,  3.0000])
```

or create a tensor basing on existing tensor. These methods will reuse properties of the input tensor, e.g. dtype, unless new values are provided by user

In [7]:

```
x = x.new_ones(5, 3, dtype=torch.double)      # new_* methods take in sizes
print(x)

x = torch.randn_like(x, dtype=torch.float)    # override dtype!
print(x)                                      # result has the same size
```

```
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]], dtype=torch.float64)
tensor([[-1.1905, -0.0900,  1.2188],
        [-0.9870,  0.0047, -0.2685],
        [-2.7798, -0.3487, -0.0234],
        [-0.4298, -0.5025,  0.4525],
        [ 1.8792, -0.5787,  1.4065]])
```

Get its size:

In [8]:

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

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

Note

``torch.Size`` is in fact a tuple, so it supports all tuple operations.

Operations ^^^^^^^^^^ There are multiple syntaxes for operations. In the following example, we will take a look at the addition operation.

Addition: syntax 1

Addition: syntax 1

In [9]:

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

```
tensor([[-0.6028,  0.2700,  2.1556],
        [-0.8627,  0.2340,  0.4082],
        [-2.4537, -0.1431,  0.2110],
        [ 0.2972,  0.0607,  0.6770],
        [ 2.3737, -0.2048,  1.8964]])
```

Addition: syntax 2

In [10]:

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

```
tensor([[-0.6028,  0.2700,  2.1556],
        [-0.8627,  0.2340,  0.4082],
        [-2.4537, -0.1431,  0.2110],
        [ 0.2972,  0.0607,  0.6770],
        [ 2.3737, -0.2048,  1.8964]])
```

Addition: providing an output tensor as argument

In [11]:

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

```
tensor([[-0.6028,  0.2700,  2.1556],
        [-0.8627,  0.2340,  0.4082],
        [-2.4537, -0.1431,  0.2110],
        [ 0.2972,  0.0607,  0.6770],
        [ 2.3737, -0.2048,  1.8964]])
```

Addition: in-place

In [12]:

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

```
tensor([[-0.6028,  0.2700,  2.1556],
        [-0.8627,  0.2340,  0.4082],
        [-2.4537, -0.1431,  0.2110],
        [ 0.2972,  0.0607,  0.6770],
        [ 2.3737, -0.2048,  1.8964]])
```

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!

In [13]:

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

```
tensor([-0.0900,  0.0047, -0.3487, -0.5025, -0.5787])
```

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

In [14]:

```
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())
```

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

If you have a one element tensor, use `.item()` to get the value as a Python number

In [15]:

```
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([ 0.5309])
0.5308841466903687
```

Read later:

100+ Tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random numbers, etc., are described `here <http://pytorch.org/docs/torch>`_.

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 a Torch Tensor to a NumPy Array ^^

In [16]:

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

```
tensor([ 1.,  1.,  1.,  1.,  1.])
```

In [17]:

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

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

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

See how the numpy array changed in value.

In [18]:

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

```
tensor([ 2.,  2.,  2.,  2.,  2.])
[2. 2. 2. 2. 2.]
```

Converting NumPy Array to Torch Tensor ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ See how changing the np array changed the Torch Tensor automatically

In [19]:

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

```
[2. 2. 2. 2. 2.]
tensor([ 2.,  2.,  2.,  2.,  2.], dtype=torch.float64)
```

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

CUDA Tensors

Tensors can be moved onto any device using the `.to` method.

In [20]:

```
# let us run this cell only if CUDA is available
# We will use ``torch.device`` objects to move tensors in and out of GPU
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on
GPU
    x = x.to(device)                       # or just use strings
``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))       # ``.to`` can also change dtype
together!
```

```
tensor([ 1.5309], device='cuda:0')
tensor([ 1.5309], dtype=torch.float64)
```

In [ ]:

In [1]:

```
%matplotlib inline
```

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 classifies digit images:

.. figure:: /_static/img/mnist.png :alt: convnet

convnet

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`

Define the network

Let's define this network:

In [2]:

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


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 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)
```

```
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, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

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()`

In [3]:

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

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

Let try a random 32x32 input 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.

In [4]:

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

```
tensor([[ 0.0247,  0.1595,  0.0741,  0.0002, -0.0312, -0.0526, -0.0536,
         -0.0530,  0.0322, -0.0133]])
```

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

In [5]:

```
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* with support for autograd operations 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 Tensor, 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 `Tensor` operation, creates at least a single `Function` node, that connects to functions that created a `Tensor` 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 <http://pytorch.org/docs/nn.html#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:

In [6]:

```
output = net(input)
target = torch.arange(1, 11)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

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

```
tensor(38.6135)
```

Now, if you follow `loss` in the backward direction, using its `.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 Tensors in the graph that has `requres_grad=True` will have their `.grad` Tensor accumulated with the gradient.

For illustration, let us follow a few steps backward:

In [7]:

```
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
```

```
<MseLossBackward object at 0x7f6c392cae48>
<AddmmBackward object at 0x7f6c392cae48>
<ExpandBackward object at 0x7f6c6c138b70>
```

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.

In [8]:

```
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)
```

```
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([ 0.,  0.,  0.,  0.,  0.,  0.])
conv1.bias.grad after backward
tensor(1.00000e-02 *
       [-3.2287, -5.5489, -2.4091,  2.8307,  1.8655,  6.6158])
```

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 <http://pytorch.org/docs/nn>`_.

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:

.. code:: python

```
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:

In [9]:

```
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 accumulat
ed
  as explained in `Backprop`_ section.
```

In [ ]:

```

```

In [1]:

```
%matplotlib inline
```

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.

.. figure:: /_static/img/cifar10.png :alt: cifar10

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
6. Loading and normalizing CIFAR10 ^^

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

In [2]:

```
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].

In [3]:

```
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')
```

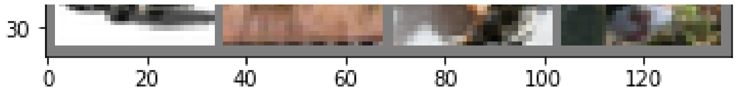
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./da
ta/cifar-10-python.tar.gz
Files already downloaded and verified
```

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

In [4]:

```
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)))
```

```
truck  deer   cat   cat
```



1. 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).

In [5]:

```
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()
```

1. Define a Loss function and optimizer ^^^ Let's use a Classification Cross-Entropy loss and SGD with momentum.

In [6]:

```
import torch.optim as optim

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

1. 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.

In [7]:

```
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

        # zero the parameter gradients
        optimizer.zero_grad()

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

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

print('Finished Training')
```

```
[1,  2000] loss: 2.145
[1,  4000] loss: 1.892
[1,  6000] loss: 1.689
[1,  8000] loss: 1.604
[1, 10000] loss: 1.526
[1, 12000] loss: 1.464
[2,  2000] loss: 1.404
[2,  4000] loss: 1.363
[2,  6000] loss: 1.343
[2,  8000] loss: 1.327
[2, 10000] loss: 1.298
[2, 12000] loss: 1.274
Finished Training
```

1. 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.

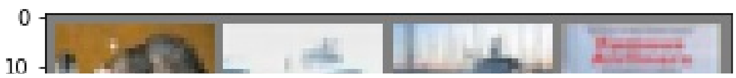
We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

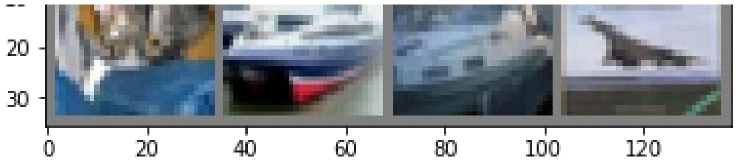
Okay, first step. Let us display an image from the test set to get familiar.

In [8]:

```
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)
))
```

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





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

In [9]:

```
outputs = net(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:

In [10]:

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

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

```
Predicted:   bird  ship   car plane
```

The results seem pretty good.

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

In [11]:

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

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

```
Accuracy of the network on the 10000 test images: 55 %
```

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:

In [12]:

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
```

```
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


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

```
Accuracy of plane : 73 %
Accuracy of   car : 78 %
Accuracy of  bird : 52 %
Accuracy of   cat : 35 %
Accuracy of  deer : 38 %
Accuracy of   dog : 32 %
Accuracy of  frog : 68 %
Accuracy of horse : 64 %
Accuracy of  ship : 59 %
Accuracy of truck : 54 %
```

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.

Let's first define our device as the first visible cuda device if we have CUDA available:

In [13]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Assume that we are on a CUDA machine, then this should print a CUDA devic
e:

print(device)
```

```
cuda:0
```

The rest of this section assumes that `device` is a CUDA device.

Then these methods will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

.. code:: python

```
    net.to(device)
```

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

.. code:: python

```
        inputs, labels = inputs.to(device), labels.to(device)
```

Why dont I notice MASSIVE speedup compared to CPU? Because your network is realllly 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 :doc:`data_parallel_tutorial`.

Where do I go next?

- :doc:`Train neural nets to play video games </intermediate/reinforcement_q_learning>`
- `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`_

In [ ]:

```

```

In [1]:

```
%matplotlib inline
```

Optional: Data Parallelism

Authors: Sung Kim <https://github.com/hunkim> *and Jenny Kang <https://github.com/jennykang>*

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:

.. code:: python

```
device = torch.device("cuda:0")
model.to(device)
```

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

.. code:: python

```
mytensor = my_tensor.to(device)
```

Please note that just calling `mytensor.to(device)` returns a new copy of `mytensor` on GPU instead of rewriting `mytensor`. You need to assign it to a new variable 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`:

.. code:: python

```
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.

In [2]:

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

# Parameters and DataLoaders
```

# Parameters and DataLoaders
input_size = 5
output_size = 2

batch_size = 30
data_size = 100
```

Device

In [3]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Dummy DataSet

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

In [4]:

```
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.

In [5]:

```
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("\tIn 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.to(device)`

In [6]:

```
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)

model.to(device)
```

Out[6]:

```
Model(
  (fc): Linear(in_features=5, out_features=2, bias=True)
)
```

Run the Model

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

In [7]:

```
for data in rand_loader:
    input = data.to(device)
    output = model(input)
    print("Outside: input size", input.size(),
          "output_size", output.size())
```

```
 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:

    code:: bash
```

.. code:: bash

```
    # 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:

.. code:: bash

```
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
```

```
    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:

.. code:: bash

```
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.

In [ ]: