Pytorch

INTRODUCTION TO PYTORCH

PyTorch is a library for processing tensors. A tensor is a number, vector, matrix or any n-dimensional array.

TENSOR

Tensor - A torch.Tensor is a multi-dimensional matrix containing elements of a single data type. It behave like the list in c++, eg - all values of tensor can only have one data type, its dimension should be symmetric means if we create a 2-D tensor then if one row has 3 elements then all rows should have 3 element otherwise it give error

A tensor can be constructed from a Python list or sequence using the torch.tensor() constructor

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

```
Import torch
# Number
t1 = torch.tensor(4)
# Vector
t2 = torch.tensor([1., 2, 3, 4])
# Matrix
t3 = torch.tensor([[5., 6],
                             [7, 8],
                             [9, 10]])
# 3-dimensional array
t4 = torch.tensor([
            [[11, 12, 13],
             [13, 14, 15]],
            [[15, 16, 17],
             [17, 18, 19.]]])
```

Tensors can have any number of dimensions, and different lengths along each dimension. We can inspect the length along each dimension using the .shape property of a tensor.

```
# to get the size of tensor
print(t2.size())
```

The contents of a tensor can be accessed and modified using Python's indexing and slicing notation:

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x[1][2])
tensor(6)
x[0][1] = 8
print(x)
tensor([[ 1,  8,  3],
        [ 4,  5,  6]])
print(x[:, 1])
tensor([4, 5, 6])
```

Tensor Functions

```
#Construct a 5x3 matrix, uninitialised
x = torch.empty(5, 3)
print(x)
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])


y = torch.zeros([2, 4], dtype=torch.int32)
print(y)
tensor([[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]], dtype=torch.int32)


Construct a randomly initialized matrix:
x = torch.rand(5, 3)
print(x)
tensor([[0.3299, 0.4388, 0.7435],
        [0.6485, 0.2348, 0.7131],
        [0.9334, 0.0560, 0.5970],
        [0.6299, 0.5537, 0.0511],
        [0.0482, 0.5627, 0.1737]])


# result y has the same size as x
y = torch.randn_like(x, dtype=torch.float)    # can override dtype!
print(y)
tensor([[ 0.7274, -1.3974,  1.1996],
        [ 0.3047, -0.1998, -0.3837],
        [-0.6805, -0.5513,  0.6728],
        [-0.4753, -0.4394,  2.1323],
        [ 1.1536,  0.9722, -0.5884]])
```

Use torch.Tensor.item() to get a Python number from a tensor containing a single value:

```
x = torch.tensor([[1]])
print(x)
tensor([[ 1]])
print(x.item())
1


x = torch.tensor(2.5)
print(x)
tensor(2.5000)
print(x.item())
2.5
```

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

We can combine tensors with the usual arithmetic operations.

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4.)
b = torch.tensor(5.)
# Arithmetic operations
y = w * x + b
```

@ represents matrix multiplication in PyTorch, and the .t method returns the transpose of a tensor.

```
y = x @ w.t() + b
```

$$\begin{array}{ccccc} X & \times & W^T & + & b \\ \begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} & \times & \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} & + & \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix} \end{array}$$

Interoperability with Numpy

Numpy is a popular open source library used for mathematical and scientific computing in Python. It enables efficient operations on large multi-dimensional arrays, and has a large ecosystem of supporting libraries:

- Matplotlib for plotting and visualization
- OpenCV for image and video processing
- Pandas for file I/O and data analysis

Instead of reinventing the wheel, PyTorch interoperates really well with Numpy to leverage its existing ecosystem of tools and libraries.

```
# Convert the numpy array to a torch tensor.
y = torch.from_numpy(x)
# Convert a torch tensor to a numpy array
z = y.numpy()
```

The interoperability between PyTorch and Numpy is really important because most datasets you'll work with will likely be read and preprocessed as Numpy arrays.

CUDA Tensors

Tensors can be moved onto any device using the .to method. Device here means Cpu or Gpu. Because u can perform computation on tensor either on gpu or cpu

```
# let us run this cell only if CUDA(GPU) 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([0.6613], device='cuda:0')
tensor([0.6613], dtype=torch.float64)
```

Autograd: Automatic Differentiation

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.

torch.Tensor is the central class of the package. If you set its attribute .requires_grad as True, it starts to track all operations on it. When you finish your computation you can call .backward() and have all the gradients computed automatically. The gradient for this tensor will be accumulated into .grad attribute.

To stop a tensor from tracking history, you can call .detach() to detach it from the computation history, and to prevent future computation from being tracked.

To prevent tracking history (and using memory), you can also wrap the code block in with torch.no_grad():. This can be particularly helpful when evaluating a model because the model may have trainable parameters with requires_grad=True, but for which we don't need the gradients.

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

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

If you want to compute the derivatives, you can call .backward() on a Tensor. If Tensor 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 gradient argument that is a tensor of matching shape.

A tensor can be created with requires_grad=True so that torch.autograd records operations on them for automatic differentiation.

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)


y = x + 2
print(y)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

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

```
print(y.grad_fn)
<AddBackward0 object at 0x7f371ea6fa58>
```

Do more operations on y

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


print(z, out)
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>) tensor(27., grad_fn=<MeanBackward0>)
```

.requires_grad_(...) changes an existing Tensor's requires_grad flag in-place. The input flag defaults to False if not given.

```
a = torch.randn(2, 2)
a = ((a * 3) / (a - 1))
print(a.requires_grad)
a.requires_grad_(True)
print(a.requires_grad)
b = (a * a).sum()
print(b.grad_fn)




False
True
<SumBackward0 object at 0x7f371ea04400>
```

Each tensor has an associated torch.Storage, which holds its data. The tensor class also provides multi-dimensional, strided view of a storage and defines numeric operations on it.

Gradients

Let's backprop now. Because out contains a single scalar, out.backward() is equivalent to out.backward(torch.tensor(1.)).

```
out.backward()
```

Print gradients d(out)/dx

```
print(x.grad)
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

You should have got a matrix of 4.5. Let’s call the out *Tensor* “*o*”. We have that *o*=14∑*izi*, *zi*=3(*xi*+2)2 and *zi*∣∣*xi*=1=27. Therefore, ∂*o*∂*xi*=32(*xi*+2), hence ∂*o*∂*xi*∣∣*xi*=1=92=4.5.

Mathematically, if you have a vector valued function $\vec{y} = f(\vec{x})$, then the gradient of $\vec{y}$ with respect to $\vec{x}$ is a Jacobian matrix:

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Generally speaking, torch.autograd is an engine for computing vector-Jacobian product. That is, given any vector *v*=(*v*1*v*2⋯*vm*)*T*, compute the product *vT*⋅*J*. If *v* happens to be the gradient of a scalar function *l*=*g*($\vec{y}$), that is, *v*=(∂*l*∂*y*1⋯∂*l*∂*ym*)*T*, then by the chain rule, the vector-Jacobian product would be the gradient of *l* with respect to $\vec{x}$:

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

This characteristic of vector-Jacobian product makes it very convenient to feed external gradients into a model that has non-scalar output.

Now let’s take a look at an example of vector-Jacobian product:

```
x = torch.randn(3, requires_grad=True)
y = x * 2
while y.data.norm() < 1000:
    y = y * 2
print(y)
tensor([  427.5083,  -866.1080, -1479.5447], grad_fn=<MulBackward0>)
```

Now in this case y is no longer a scalar. torch.autograd could not compute the full Jacobian directly, but if we just want the vector-Jacobian product, simply pass the vector to backward as argument:

```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float)
```

```
y.backward(v)
print(x.grad)


tensor([5.1200e+01, 5.1200e+02, 5.1200e-02])
```

You can also stop autograd from tracking history on Tensors with .requires_grad=True either by wrapping the code block in with torch.no_grad() Or by using .detach() to get a new Tensor with the same content but that does not require gradients:

we can automatically compute the derivative of y w.r.t. the tensors that have requires_grad set to True i.e. w and b. To compute the derivatives, we can call the .backward method on our result y.

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4.,requires_grad=True)
b = torch.tensor(5.,requires_grad=True)
# Arithmetic operations
y = w * x + b
# Compute derivatives
y.backward()


# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)


dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(3.)
```

As expected, dy/dw has the same value as x i.e. 3, and dy/db has the value 1. Note that x.grad is None, because x doesn't have requires_grad set to True.

The "grad" in w.grad stands for gradient, which is another term for derivative, used mainly when dealing with matrices.

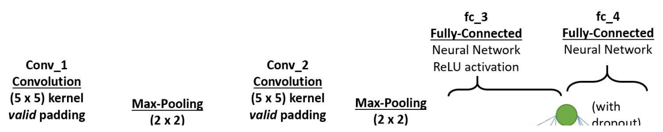
u can check weather variable has grad or not

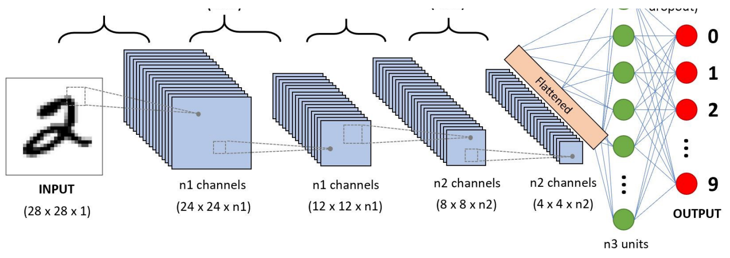
```
print(w.grad)
True
```

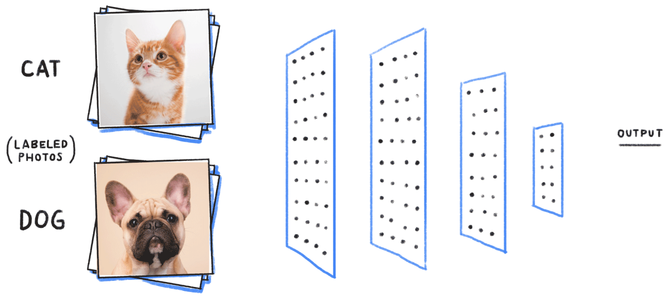
Neural Networks

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.







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

Defining a Neural Network in PyTorch

Deep learning uses artificial neural networks (models), which are computing systems that are composed of many layers of interconnected units. By passing data through these interconnected units, a neural network is able to learn how to approximate the computations required to transform inputs into outputs. In PyTorch, neural networks can be constructed using the torch.nn package.

Introduction

PyTorch provides the elegantly designed modules and classes, including torch.nn, to help you create and train neural networks. An nn.Module contains layers, and a method forward(input) that returns the output.

In this recipe, we will use torch.nn to define a neural network intended for the MNIST dataset.

Steps

1. Import all necessary libraries for loading our data
2. Define and intialize the neural network

3. Specify how data will pass through your model
4. Pass data through your model to test

1. Import necessary libraries for loading our data

For this recipe, we will use torch and its subsidiaries torch.nn and torch.nn.functional.

import torch

import torch.nn as nn

import torch.nn.functional as F

2. Define and intialize the neural network

Our network will recognize images. We will use a process built into PyTorch called convolution. Convolution adds each element of an image to its local neighbors, weighted by a kernel, or a small martrix, that helps us extract certain features (like edge detection, sharpness, blurriness, etc.) from the input image.
There are two requirements for defining the Net class of your model. The first is writing an __init__ function that references nn.Module. This function is where you define the fully connected layers in your neural network.
Using convolution, we will define our model to take 1 input image channel, and output match our target of 10 labels representing numbers 0 through 9. This algorithm is yours to create, we will follow a standard MNIST algorithm.

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




      # First 2D convolutional layer, taking in 1 input channel (image),
      # outputting 32 convolutional features, with a square kernel size of 3
      self.conv1 = nn.Conv2d(1, 32, 3, 1)
      # Second 2D convolutional layer, taking in the 32 input layers,
      # outputting 64 convolutional features, with a square kernel size of 3
      self.conv2 = nn.Conv2d(32, 64, 3, 1)




      # Designed to ensure that adjacent pixels are either all 0s or all active
      # with an input probability
      self.dropout1 = nn.Dropout2d(0.25)
      self.dropout2 = nn.Dropout2d(0.5)




      # First fully connected layer
      self.fc1 = nn.Linear(9216, 128)
      # Second fully connected layer that outputs our 10 labels
      self.fc2 = nn.Linear(128, 10)
```

```
        self.fc2 = nn.Linear(128, 10)



my_nn = Net()
print(my_nn)
```

We have finished defining our neural network, now we have to define how our data will pass through it.

3. Specify how data will pass through your model

When you use PyTorch to build a model, you just have to define the forward function, that will pass the data into the computation graph (i.e. our neural network). This will represent our feed-forward algorithm.

You can use any of the Tensor operations in the forward function.

```
class Net(nn.Module):
    def __init__(self):
      super(Net, self).__init__()
      self.conv1 = nn.Conv2d(1, 32, 3, 1)
      self.conv2 = nn.Conv2d(32, 64, 3, 1)
      self.dropout1 = nn.Dropout2d(0.25)
      self.dropout2 = nn.Dropout2d(0.5)
      self.fc1 = nn.Linear(9216, 128)
      self.fc2 = nn.Linear(128, 10)


    # x represents our data
    def forward(self, x):
      # Pass data through conv1
      x = self.conv1(x)
      # Use the rectified-linear activation function over x
      x = F.relu(x)
      x = self.conv2(x)
      x = F.relu(x)


      # Run max pooling over x
      x = F.max_pool2d(x, 2)
      # Pass data through dropout1
      x = self.dropout1(x)
      # Flatten x with start_dim=1
      x = torch.flatten(x, 1)
      # Pass data through fc1
      x = self.fc1(x)
      x = F.relu(x)
      x = self.dropout2(x)
      x = self.fc2(x)
```

```
        # Apply softmax to x
        output = F.log_softmax(x, dim=1)
        return output
```

4. Pass data through your model to test

To ensure we receive our desired output, let's test our model by passing some random data through it.

```
# Equates to one random 28x28 image
random_data = torch.rand((1, 1, 28, 28))


my_nn = Net()
result = my_nn(random_data)
print (result)
```

Each number in this resulting tensor equates to the prediction of the label the random tensor is associated to.

Example of full neural network

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        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=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, 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 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


10
torch.Size([6, 1, 3, 3])
```

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.

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


loss = criterion(output, target)
```

```
print(loss)


tensor(1.3319, grad_fn=<MseLossBackward>)
```

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 requires_grad=True will have their .grad Tensor 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


<MseLossBackward object at 0x7f7a8434da20>
<AddmmBackward object at 0x7f7a8434da90>
<AccumulateGrad object at 0x7f7a8434da90>
```

BackpropTo 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)


conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([ 0.0165, -0.0092,  0.0073, -0.0145, -0.0016,  0.0088])
```

Now, we have seen how to use loss functions.

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
```

Training on GPU

Just like how you transfer a Tensor onto 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:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# Assuming that we are on a CUDA machine, this should print a CUDA device:
print(device)


cuda:0
```

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