Jonathan Hui blog About

“PyTorch - Neural networks with nn modules”

Feb 9, 2018

The *nn* modules in PyTorch provides us a higher level API to build and train deep network.

Neural Networks

In PyTorch, we use torch.nn to build layers. For example, in *__iniit__*, we configure different trainable layers including convolution and affine layers with *nn.Conv2d* and *nn.Linear* respectively. We create the method *forward* to compute the network output. It contains functionals linking layers already configured in *__iniit__* to form a computation graph. Functionals include ReLU and max poolings.

To create a deep 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 co
        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))

        # 2 is ame as (2, 2)
        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 di
        num_features = 1
        for s in size:       # Get the products
            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)
#  (fc2): Linear(in_features=120, out_features=84)
#  (fc3): Linear(in_features=84, out_features=10)
#)
```

The learnable parameters of a model are returned by *net.parameters*. For example, *params*[0] returns the trainable parameters for conv1 which has the size of 6x1x5x5.

```
params = list(net.parameters())
```

```
print(len(params))       # 10: 10 sets of trainable parameters

print(params[0].size())  # torch.Size([6, 1, 5, 5])
```

We compute the network output by:

```
input = Variable(torch.randn(1, 1, 32, 32))
out = net(input)   # out's size: 1x10.
# Variable containing:
# 0.1268  0.0207  0.0857  0.1454 -0.0370  0.0030  0.0150 -0.0542
# [torch.FloatTensor of size 1x10]
```

input here has a size of (batch size) x (# of channel) x width x height. torch.nn processes batch data only. To support a single datapoint, use *input.unsqueeze(0)* to convert a single datapoint to a batch with only one sample.

Net extends from *nn.Module*. Hence, *Net* is a reusable custom module just like other built-in modules (layers) provided by *nn.Module*.

Variables and functional

The difference between *torch.nn* and *torch.nn.functional* is very subtle. In fact, many *torch.nn.functional* have a corresponding equivalent in *torch.nn*. For layers with trainable parameters, we use *torch.nn* to create the layer. We store it back in the instance so we can easily access the layer and the trainable parameters later.

```
self.conv1 = nn.Conv2d(1, 6, 5)
```

In many code samples, it uses *torch.nn.functional* for simpler operations that have no trainable parameters or configurable parameters. Alternatively, in a later section, we use *torch.nn.Sequential* to compose layers from *torch.nn* only. Both approaches are simple and more like a coding style issue rather than any major implementation differences.

Common Functionals

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

data = autograd.Variable(torch.randn(2, 2))
F.relu(data)

data = autograd.Variable(torch.randn(5))
F.softmax(data, dim=0)
```

Backward pass

To compute the backward pass for gradient, we first zero the gradient stored in the network. In PyTorch, every time we backpropagate the gradient from a variable, the gradient is accumulative instead of being reset and replaced. In some network designs, we need to call *backward* multiple times. For example in a generative adversary network GAN, we need an accumulated gradients from 2 *backward* passes: one for the generative part and one for the adversary part of the network. We reset the gradients only once but not between *backward* calls. Hence, to accommodate such flexibility, we explicitly reset the gradient instead of having *backward* resets it automatically every time.

```
net.zero_grad()
out.backward()
```

Loss function

PyTorch comes with many loss functions. For example, the code below create a mean square error loss function and later backpropagate the gradients based on the loss.

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

# Compute the loss by MSE of the output and the true label
loss = criterion(output, target)         # Size 1
```

```
net.zero_grad()      # zeroes the gradient buffers of all paramet

loss.backward()

# Print the gradient for the bias parameters of the first convolu
print(net.conv1.bias.grad)

# Variable containing:
# -0.0007
# -0.0400
# 0.0184
# 0.1273
# -0.0080
# 0.0387
# [torch.FloatTensor of size 6]
```

If we follow loss in the backward direction using *grad_fn* attribute, we can see a graph of computations similar to:

```
nput -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

Here are some simple print out of the function chain.

```
loss = criterion(output, target)         # Size 1

print(loss.grad_fn)                      # <MseLossBackward objec
print(loss.grad_fn.next_functions[0][0]) # <AddmmBackward object
print(loss.grad_fn.next_functions[0][0].next_functions[0][0])  #
```

Optimizer

We seldom access the gradients manually to train the model parameters. PyTorch provides *torch.optim* for such purpose. To train the parameters, we create an optimizer and call *step* to upgrade the parameters.

```
import torch.optim as optim

# Create a SGD optimizer for gradient descent
optimizer = optim.SGD(net.parameters(), lr=0.01)

# Inside the training loop
for t in range(500):
   output = net(input)
   loss = criterion(output, target)

   optimizer.zero_grad()   # zero the gradient buffers
   loss.backward()

   optimizer.step()        # Perform the training parameters upda
```

We need to zero the gradient buffer once for every training iteration to reset the gradient computed by last data batch.

Adam optimizer

Adam optimizer is one of the most popular gradient descent optimizer in deep learning. Here is the partial sample code in using an Adam optimizer:

```
learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate
```

Putting it together

To put everything together, we creats a CNN classifier for the CIFAR10 images.

Reading Dataset (torchvision)

PyTorch provides a package called *torchvision* to load and prepare dataset. First, we use *transforms.Compose* to compose a series of transformation. torchvision reads datasets into PILImage (Python imaging format). *transforms.ToTensor* converts a PIL Image in the range [0, 255] to a torch.FloatTensor of shape (C x H x W) with range [0.0, 1.0]. We then renormalize the input to [-1, 1]:

$$input = \frac{input - 0.5}{0.5}$$

torchvision.datasets.CIFAR10 is responsible for loading and transforming a dataset (training or testing). *torchvision.datasets.CIFAR10* is passed to a *torch.utils.data.DataLoader* to load multiple samples in parallel.

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


transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True
                                        download=True, transform=
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_worke

testset = torchvision.datasets.CIFAR10(root='./data', train=False
                                       download=True, transform=t
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_worke

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Model & Training

We define a model in the class *Net*. Then we run 2 epoch of training using cross entropy loss function with a SGD optimizer.

```
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()

import torch.optim as optim

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

for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    # With a batch size of 4 in each iteration
    for i, data in enumerate(trainloader, 0):  # trainloader read
        inputs, labels = data
        inputs, labels = Variable(inputs), Variable(labels)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        print(f"Loss {loss.data[0]}")

print('Finished Training')
```

Testing

To compute the accuracy for the testing data:

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

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

CUDA

To run the code in multiple GPUs:

Move the model to GPU:

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

Move all tensors to GPU:

```
if torch.cuda.is_available():
    input_var = Variable(data.cuda())
```

Calling data.cuda() won’t copy the tensor to the GPU. We need to assign it to a new tensor and use that tensor on the GPU.

PyTorch uses only one GPU by default. The steps above only run the code in one GPU. For multiple GPUs we need to run the model run in parallell with DataParallel:

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

Here is the full source code for reference:

```
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


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)


class Model(nn.Module):

    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
```

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

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


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())
```

torch.nn.Sequential

We can build a model using a Sequential without *torch.nn.functional*. Most *torch.nn.functional* classes have an equivalent in *torch.nn*. Using *torch.nn* alone or mixing it with *torch.nn.functional* are both very common. I will let the judgement to the reader.

```
import torch
from torch.autograd import Variable

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

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

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)

learning_rate = 1e-4
for t in range(500):
    y_pred = model(x)

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

    model.zero_grad()

    loss.backward()

    for param in model.parameters():
        param.data -= learning_rate * param.grad.data
```

Dynamic computation graph example

PyTorch uses a new graph for each training iteration. This allows us to have a different graph for each iteration. The code below is a fully-connected ReLU network that each forward pass has somewhere between 1 to 4 hidden layers. It also demonstrate how to share and reuse weights.

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


class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        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):
        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, D_in, H, D_out = 64, 1000, 100, 10

x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)

model = DynamicNet(D_in, H, D_out)

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum
for t in range(500):
    y_pred = model(x)

    loss = criterion(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Transfer model

In computer vision, training a model from scratch using your own dataset is time consuming. In a transfer model, we load a pre-trained model that is trained with a well known dataset. We replace the final layers with our own layers. Then we retrain the model with our own dataset. Such strategy allows us to start the retraining with good quality model parameters rather than random values. That usually results in much shorter training time to achieve the desirable accuracy. This approach often requires less training data. In some cases, we even freeze all the model parameters except the layers that we replaced, this can further cut down the training time.

Here is some boiler plate code in loading data. You should already familiar with it by now.

```
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 time
import os
import copy

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomSizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224
    ]),
    'val': transforms.Compose([
        transforms.Scale(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224
    ]),
}

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

Now, we are loading a pre-trained ResNet model using the torchvision. The last layer of the RestNet is a fully connected layer. We find out the input size of that layer and replace it with a new fully connected layer *nn.Linear* that output 2 channels . Then we will retrain a new model.

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

criterion = nn.CrossEntropyLoss()

optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentu
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,

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

train_model retrains our new model. For each epoch, we run both the training and validation data. We keep track of the best model with the highest validation accuracy and return the best model after 25 epochs.

```
def train_model(model, criterion, optimizer, scheduler, num_epoch
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        # 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
                inputs, labels = Variable(inputs), Variable(label

                # 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.dat

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

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

    time_elapsed = time.time() - since

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

As mentioned before, we can even save more training time by just retraining the replaced layer. We set *requires_grad* to False for the original model. Then we replace it with a new FC layer using *nn.Linear*. By default, a newly constructed modules have *requires_grad* =True by default.

```
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
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)

criterion = nn.CrossEntropyLoss()

optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001,
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=

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

3 Comments jhui 1 Login

Recommend 1 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Farzad Sharif • 6 months ago

Fantastic fantastic post.

• Reply • Share ›

ldtbor • 10 months ago

Could you check you my error? I cannot run your loss function

tensor(1.00000e-02 *
[[9.3416, 4.0127, 1.3793, 2.7767, -6.1698, -6.0204, -3.8576,
8.2983, 0.7076, -3.7413]])
tensor([1., 2., 3., 4., 5., 6., 7., 8., 9., 10.])
MSELoss()

```
---------------------------------------------------------------------------
RuntimeError Traceback (most recent call last)
<ipython-input-21-f37c6e3740f6> in <module>()
27
28 # Compute the loss by MSE of the output and the true label
---> 29 loss = criterion(output, target) # Size 1
30
31 net.zero_grad() # zeroes the gradient buffers of all parameters

/usr/local/lib/python3.6/dist-packages/torch/nn/modules/module.py in
__call__(self, *input, **kwargs)
```

see more

• Reply • Share ›

Jonathan Hui Mod → ldtbor • 10 months ago

I rerun my code. I don't see any error with the loss function. Sorry, you are on your own now.

• Reply • Share ›

ALSO ON JHUI

Jonathan Hui blog jhui Deep learning