PyTorch Tutorials

PyTorch

0.4.0

Search docs

BEGINNER TUTORIALS

Deep Learning with PyTorch: A 60 Minute Blitz
PyTorch for former Torch users
Learning PyTorch with Examples
Tensors
Autograd
nn module
Examples
Tensors
Autograd
nn module
PyTorch: nn
PyTorch: optim
PyTorch: Custom nn Modules
PyTorch: Control Flow + Weight Sharing
Transfer Learning tutorial
Data Loading and Processing Tutorial
Deep Learning for NLP with Pytorch

INTERMEDIATE TUTORIALS

Classifying Names with a Character-Level RNN
Generating Names with a Character-Level RNN
Translation with a Sequence to Sequence Network and Attention

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.

```python
import random
import torch


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

# 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.item())

    # 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)

Download Python source code: dynamic_net.py

Download Jupyter notebook: dynamic_net.ipynb

Gallery generated by Sphinx-Gallery

Previous Next