fit_sine_with_third_order_polynomial

March 2, 2021

1 Fit $y = sin(x)$ with a third order polynomial

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

1.1 nn package

[1]:
```
import torch
import math
```

[2]:
```
# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)
print(f"x= {x}\n")
print(f"y= {y}\n")
print(x.shape, y.shape)
```

```
x= tensor([-3.1416, -3.1384, -3.1353,  …,  3.1353,  3.1384,  3.1416])

y= tensor([ 8.7423e-08, -3.1430e-03, -6.2863e-03,  …,  6.2863e-03,
         3.1432e-03, -8.7423e-08])

torch.Size([2000]) torch.Size([2000])
```

For this example, the output y is a linear function of (x, x^2, x^3), so we can consider it as a linear layer neural network. Let's prepare the tensor (x, x^2, x^3).

[3]:
```
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)
print(f"x.unsqueeze(-1) =\n {x.unsqueeze(-1)}\n")
print(f"x.unsqueeze(-1).shape = {x.unsqueeze(-1).shape}\n")
print(f"xx = {xx}\n")
print(f"xx.shape = {xx.shape}\n")
```

```
x.unsqueeze(-1) =
 tensor([[-3.1416],
        [-3.1384],
        [-3.1353],
        …,
        [ 3.1353],
        [ 3.1384],
        [ 3.1416]])

x.unsqueeze(-1).shape = torch.Size([2000, 1])

xx = tensor([[ -3.1416,   9.8696, -31.0063],
        [ -3.1384,   9.8499, -30.9133],
        [ -3.1353,   9.8301, -30.8205],
        …,
        [  3.1353,   9.8301,  30.8205],
        [  3.1384,   9.8499,  30.9133],
        [  3.1416,   9.8696,  31.0063]])

xx.shape = torch.Size([2000, 3])
```

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. The Linear Module computes output from input using a linear function, and holds internal Tensors for its weight and bias.
The Flatten layer flatens the output of the linear layer to a 1D tensor, to match the shape of `y`.

[4]:
```
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)# Flattens a contiguous range of dims into a tensor. ␣
 ↪                                    torch.nn.Flatten(start_dim: int = 1,␣
 ↪end_dim: int = -1)
)
```

The nn package also contains definitions of popular loss functions; in this case we will use Mean Squared Error (MSE) as our loss function.

[5]:
```
loss_fn = torch.nn.MSELoss(reduction='sum')
```

[6]:
```
learning_rate = 1e-6
```

[7]:
```
for t in range(3000):

    # 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 Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(xx)
    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 200 == 199:
        print(f"{t+1}-th epoch: MSE = {loss.item()}") # '.item()' is to obtain␣
↪its value

    # 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 Tensors 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 Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

```
200-th epoch: MSE = 420.2522888183594
400-th epoch: MSE = 194.13136291503906
600-th epoch: MSE = 92.5570068359375
800-th epoch: MSE = 46.79005432128906
1000-th epoch: MSE = 26.10063934326172
1200-th epoch: MSE = 16.714818954467773
1400-th epoch: MSE = 12.440919876098633
1600-th epoch: MSE = 10.487028121948242
1800-th epoch: MSE = 9.590081214904785
2000-th epoch: MSE = 9.176529884338379
2200-th epoch: MSE = 8.98501968383789
2400-th epoch: MSE = 8.895927429199219
2600-th epoch: MSE = 8.854293823242188
2800-th epoch: MSE = 8.834747314453125
3000-th epoch: MSE = 8.82552719116211
```

[8]:
```
# You can access the first layer of `model` like accessing the first item of a␣
↪list
linear_layer = model[0]
print(linear_layer)
print(linear_layer.bias)
print(linear_layer.weight)
```

```
Linear(in_features=3, out_features=1, bias=True)
Parameter containing:
tensor([-0.0024], requires_grad=True)
Parameter containing:
tensor([[ 8.5504e-01,  4.2100e-04, -9.3089e-02]], requires_grad=True)
```

[9]:
```
# For linear layer, its parameters are stored as `weight` and `bias`.
print(f'The third order polynomial aproximation of sine function is :\n\ty =␣
↪{linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x +␣
↪{linear_layer.weight[:, 1].item()} x^2 + {linear_layer.weight[:, 2].item()}␣
↪x^3')
```

```
The third order polynomial aproximation of sine function is :
        y = -0.0024403163697570562 + 0.8550443649291992 x +
0.00042099523125216365 x^2 + -0.09308907389640808 x^3
```

1.2 Optim package

Up to this point we have updated the weights of our models by manually mutating the Tensors holding learnable parameters with `torch.no_grad()`. 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 RMSprop algorithm provided by the optim package:

[10]:
```
import torch
import math


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Prepare the input tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss(reduction='sum')
```

[11]:
```
# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use RMSprop; the optim package contains many␣
↪other
# optimization algorithms. The first argument to the RMSprop constructor tells␣
↪the
# optimizer which Tensors it should update.
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
# optimizer = torch.optim.Adagrad(model.parameters(), lr=learning_rate)
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

[12]:
```
for t in range(3000):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 200 == 199:
        print(f"{t+1}-th epoch: MSE = {loss.item()}")

    # 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()


linear_layer = model[0]
print(f'The third order polynomial aproximation of sine function is :\n\ty =␣
↪{linear_layer.bias.item()} + {linear_layer.weight[:, 0].item()} x +␣
↪{linear_layer.weight[:, 1].item()} x^2 + {linear_layer.weight[:, 2].item()}␣
↪x^3')
```

```
200-th epoch: MSE = 143.6226348876953
400-th epoch: MSE = 54.95341491699219
```

```
600-th epoch: MSE = 15.61046314239502
800-th epoch: MSE = 8.947502136230469
1000-th epoch: MSE = 8.973044395446777
1200-th epoch: MSE = 8.841585159301758
1400-th epoch: MSE = 8.927045822143555
1600-th epoch: MSE = 8.915283203125
1800-th epoch: MSE = 8.92263126373291
2000-th epoch: MSE = 8.91980266571045
2200-th epoch: MSE = 8.921185493469238
2400-th epoch: MSE = 8.920588493347168
2600-th epoch: MSE = 8.920843124389648
2800-th epoch: MSE = 8.92072868347168
3000-th epoch: MSE = 8.920766830444336
The third order polynomial aproximation of sine function is :
        y = 0.0004999875091016293 + 0.8562408089637756 x + 0.0004999732482247055
x^2 + -0.09383039176464081 x^3
```