Politecnico di Torino

1859

PyTorch Bootcamp

Machine Learning and Deep Learning course, A. A. 2024/25

Credits: *Simone Alberto Peirone, Arda Eren Dogru*
Slide credits: *Chiara Plizzari*

Training a Custom Model

Lesson 2

torch.nn.Module

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let's check to see if torch.cuda is available, else we continue to use the CPU

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Using {} device'.format(device))
```

Out:

```
Using cuda device
```

Define the class

We define our neural network by subclassing nn.Module, and initialize the neural network layers in __init__. Every nn.Module subclass implements the operations on input data in the forward method.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Define the class

We create an instance of NeuralNetwork, and move it to the device, and print its structure

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
    (5): ReLU()
  )
)
```

Define the class

To use the model, we pass it the input data. This executes the model's forward, along with some background operations. Do not call model.forward() directly!

Calling the model on the input returns a 10-dimensional tensor with raw predicted values for each class. We get the prediction probabilities by passing it through an instance of the nn.Softmax module.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([2], device='cuda:0')
```

Loss function

Common loss functions include nn.MSELoss (Mean Square Error) for regression tasks, and nn.NLLLoss (Negative Log Likelihood) for classification. nn.CrossEntropyLoss combines nn.LogSoftmax and nn.NLLLoss.

We pass our model's output logits to nn.CrossEntropyLoss, which will normalize the logits and compute the prediction error.

```
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

Optimizer

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

- Call optimizer.zero_grad() to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to loss.backward(). PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call optimizer.step() to adjust the parameters by the gradients collected in the backward pass.

Autograd

- Automatic Differentiation Package
- Don’t need to worry about partial differentiation, chain rule etc.
    - `backward()` does that
- Gradients are accumulated for each step by default:
    - Need to zero out gradients after each update
    - `tensor.grad_zero()`

Full Implementation - Train Loop

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Full Implementation - Test Loop

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= size
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

Full Implementation

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

Full Implementation

Out:

```
Epoch 1
-------------------------------
loss: 2.299511  [    0/60000]
loss: 2.301767  [ 6400/60000]
loss: 2.289777  [12800/60000]
loss: 2.291731  [19200/60000]
loss: 2.269755  [25600/60000]
loss: 2.261175  [32000/60000]
loss: 2.258553  [38400/60000]
loss: 2.240743  [44800/60000]
loss: 2.260818  [51200/60000]
loss: 2.243683  [57600/60000]
Test Error:
 Accuracy: 37.3%, Avg loss: 0.035121

Epoch 2
-------------------------------
loss: 2.229830  [    0/60000]
loss: 2.241497  [ 6400/60000]
loss: 2.221580  [12800/60000]
```

Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called state_dict. These can be persisted via the torch.save method:

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using load_state_dict() method.

```
model = models.vgg16() # we do not specify pretrained=True, i.e. do not load default weights
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

Your turn! 

It’s time to implement your custom Neural Network!

- At this link http://cs231n.stanford.edu/tiny-imagenet-200.zip you can find the TinyImageNet dataset.
- Implement your Neural Network for classification
- Implement your training and test loop
- Which training accuracy can your network achieve?
- Which test accuracy can your network achieve?

Notebook @
https://colab.research.google.com/drive/1AyPjSbUWr6Y46mlYNqwRA7BO3Yuxwo1m