Colab Notebook GitHub

Optimizing Model Parameters

Created On: Feb 09, 2021 | Last Updated: Apr 28, 2025 | Last Verified: Nov 05, 2024

Now that we have a model and data it's time to train, validate and test our model by optimizing its parameters on our data. Training a model is an iterative process; in each iteration the model makes a guess about the output, calculates the error in its guess (*loss*), collects the derivatives of the error with respect to its parameters (as we saw in the previous section), and **optimizes** these parameters using gradient descent. For a more detailed walkthrough of this process, check out this video on backpropagation from 3Blue1Brown.

Prerequisite Code

We load the code from the previous sections on Datasets & DataLoaders and Build Model.

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

training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__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),
        )

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

model = NeuralNetwork()
```

Out:

```
  0%|          | 0.00/26.4M [00:00<?, ?B/s]
  0%|          | 65.5k/26.4M [00:00<01:12, 363kB/s]
  1%|          | 229k/26.4M [00:00<00:38, 681kB/s]
  3%|▎         | 918k/26.4M [00:00<00:12, 2.10MB/s]
 14%|█▍        | 3.67M/26.4M [00:00<00:03, 7.25MB/s]
 37%|███▋      | 9.70M/26.4M [00:00<00:01, 16.6MB/s]
 60%|██████    | 15.7M/26.4M [00:01<00:00, 22.2MB/s]
 82%|████████▏ | 21.8M/26.4M [00:01<00:00, 25.8MB/s]
100%|██████████| 26.4M/26.4M [00:01<00:00, 19.3MB/s]

  0%|          | 0.00/29.5k [00:00<?, ?B/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 328kB/s]

  0%|          | 0.00/4.42M [00:00<?, ?B/s]
  1%|▏         | 65.5k/4.42M [00:00<00:12, 363kB/s]
  5%|▌         | 229k/4.42M [00:00<00:06, 682kB/s]
 21%|██▏       | 918k/4.42M [00:00<00:01, 2.11MB/s]
 83%|████████▎ | 3.67M/4.42M [00:00<00:00, 7.28MB/s]
100%|██████████| 4.42M/4.42M [00:00<00:00, 6.10MB/s]

  0%|          | 0.00/5.15k [00:00<?, ?B/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 49.1MB/s]
```

Hyperparameters

Hyperparameters are adjustable parameters that let you control the model optimization process. Different hyperparameter values can impact model training and convergence rates (read more about hyperparameter tuning)

We define the following hyperparameters for training:

- **Number of Epochs** - the number of times to iterate over the dataset
- **Batch Size** - the number of data samples propagated through the network before the parameters are updated
- **Learning Rate** - how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

Optimization Loop

Once we set our hyperparameters, we can then train and optimize our model with an optimization loop. Each iteration of the optimization loop is called an **epoch**.

Each epoch consists of two main parts:

- **The Train Loop** - iterate over the training dataset and try to converge to optimal parameters.
- **The Validation/Test Loop** - iterate over the test dataset to check if model performance is improving.

Let's briefly familiarize ourselves with some of the concepts used in the training loop. Jump ahead to see the Full Implementation of the optimization loop.

Loss Function

When presented with some training data, our untrained network is likely not to give the correct answer. **Loss function** measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training. To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value.

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

Optimization is the process of adjusting model parameters to reduce model error in each training step. **Optimization algorithms** define how this process is performed (in this example we use Stochastic Gradient Descent). All optimization logic is encapsulated in the `optimizer` object.

Here, we use the SGD optimizer; additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.

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)
```

Inside the training loop, optimization happens in three steps:

- 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.

Full Implementation

We define `train_loop` that loops over our optimization code, and `test_loop` that evaluates the model's performance against our test data.

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and drop
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

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

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


def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dr
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are com
    # also serves to reduce unnecessary gradient computations and memory usage fo
    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 /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss
```


We initialize the loss function and optimizer, and pass it to `train_loop` and `test_loop`. Feel free to increase the number of epochs to track the model's improving performance.

Tutorials

Get in-depth tutorials for beginners and advanced developers

View Tutorials

Resources

Find development resources and get your questions answered

View Resources