Introduction to Deep Learning (CS474)

Lecture 8

Rupam

Outline

- Mechanics of Learning-PART III
    - Optimizers

Introduction

- Vanilla **gradient descent** for optimization was shown to you in earlier classes, which worked fine for our simple case.

- Needless to say, there are several optimization strategies and tricks that can assist convergence, especially when models get complicated.

- Now, it is the right time to introduce the way **PyTorch** abstracts the optimization strategy away from user code: that is, the training loop we've examined.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Introduction

- The **torch module** has an **optim** submodule where we can find classes implementing different optimization algorithms.
- Before exercising this module, I hope that you remember our last *notebook* example.

```python
import numpy as np
import torch

t_c = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0,
                    8.0, 3.0, -4.0, 6.0, 13.0, 21.0])
t_u = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9,
                    33.9, 21.8, 48.4, 60.4, 68.4])
t_un = 0.1 * t_u

def model(t_u, w, b):
    return w * t_u + b

def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Introduction

- Continuing with the last *notebook*!

```
import torch.optim as optim

dir(optim)
```

```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'AdamW',
 'Adamax',
 'LBFGS',
 'Optimizer',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Conceptual Representation of an Optimizer

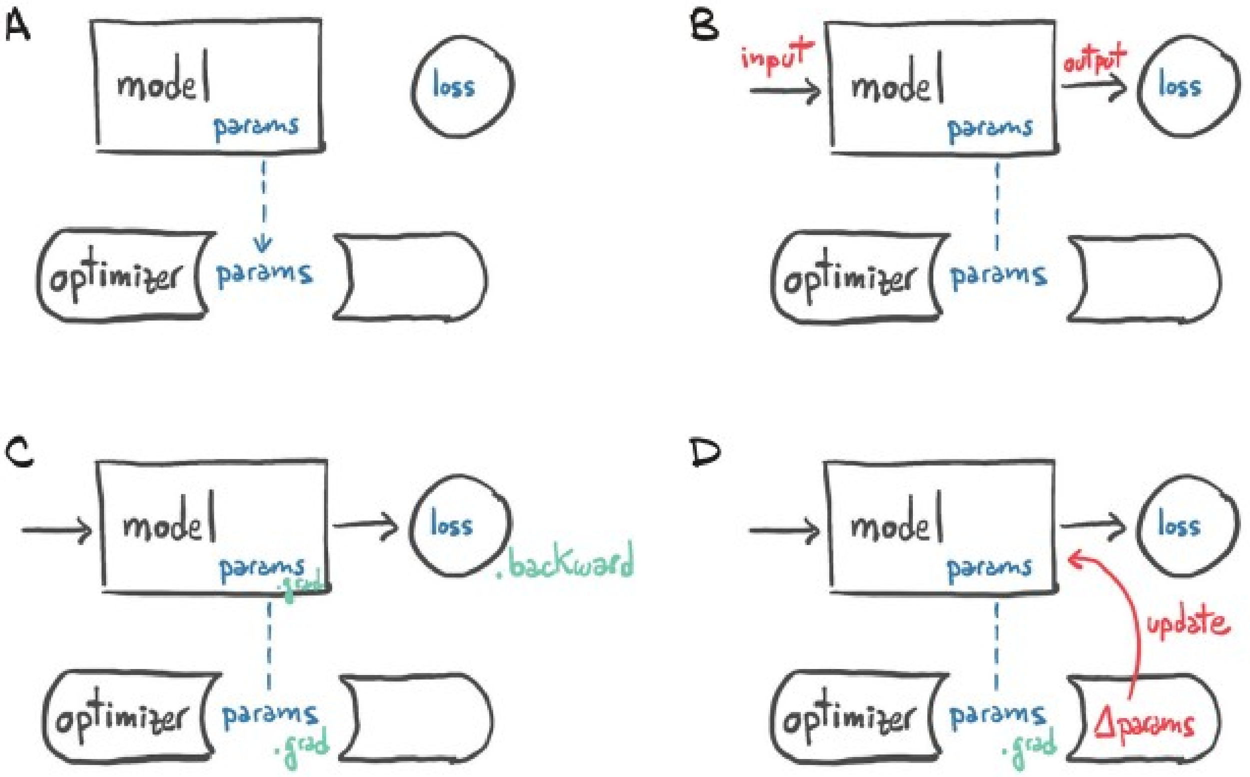

Figure 1

Image credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Conceptual Representation of an Optimizer

Look into the **figure 1**!

(A) Conceptual representation of how an optimizer holds a reference to parameters.

(B) After a loss is computed from inputs,

(C) a call to `.backward` leads to .grad being populated on parameters.

(D) At that point, the optimizer can access `.grad` and compute the parameter updates.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Applying SGD Optimizer through PyTorch

- Here **SGD** stands for stochastic gradient descent.

- The term **stochastic** comes from the fact that the gradient is typically obtained by averaging over a random subset of all input samples, called a **minibatch.**

- However, the optimizer does not know if the loss was evaluated on all the samples (vanilla) or a random subset of them (stochastic), so the algorithm is literally the same in the two cases.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Applying SGD Optimizer through PyTorch

- Continuing with the earlier *notebook.*

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)

optimizer.zero_grad() # <1>
loss.backward()
optimizer.step()

params
```

```
tensor([1.7761, 0.1064], requires_grad=True)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Applying SGD Optimizer through PyTorch

- Continuing with the earlier *notebook.*

```
# Updated training loop!
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

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

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Applying SGD Optimizer through PyTorch

- Continuing with the earlier *notebook.*

```
# Invoking training loop

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

```
Epoch 500, Loss 7.843371
Epoch 1000, Loss 3.825484
Epoch 1500, Loss 3.091631
Epoch 2000, Loss 2.957596
Epoch 2500, Loss 2.933116
Epoch 3000, Loss 2.928646
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927680
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647
tensor([  5.3671, -17.3012], requires_grad=True)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Applying SGD Optimizer through PyTorch

- The optim module helps us abstract away the specific optimization scheme.

- All we have to do is provide a list of params to it (that list can be extremely, as is needed for very deep neural network models), and we can forget about the details.

- In order to test more optimizers, all we have to do is instantiate a different optimizer, say Adam , instead of SGD . The rest of the code stays as it is. Pretty handy stuff.

- We won't go into much detail about Adam; suffice to say that it is a more sophisticated optimizer in which the learning rate is set adaptively.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Applying SGD Optimizer through PyTorch

- We have touched on a lot of the essential concepts that will enable us to train complicated deep learning models while knowing what's going on under the hood:
  - backpropagation to estimate gradients,
  - **autograd**, and
  - **optimizing** weights of models
  - using gradient descent or other optimizers.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

References

- All the contents present in the slides are taken from various online resources. Due credit is given in the respective slides.These slides are used for *academic* purposes only.