

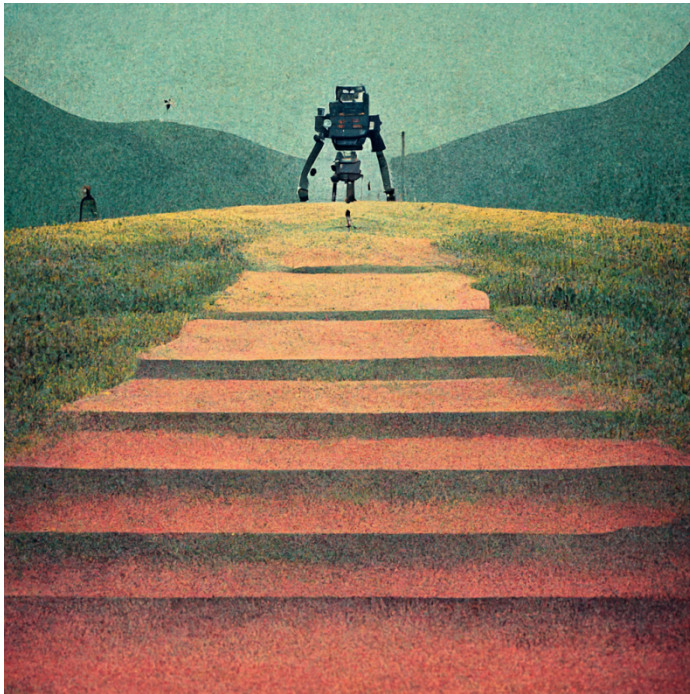
CS197 Harvard: AI Research Experiences

Fall 2022: Lectures 6 & 7 – “Moonwalking with PyTorch”
Solidifying PyTorch Fundamentals

Instructed by Pranav Rajpurkar. Website <https://cs197.seas.harvard.edu/>

Abstract

Last lecture, in our coding walkthrough, we saw how PyTorch was being used within a codebase, but we did not dive into the fundamentals of PyTorch. Today, we will solidify our understanding of the PyTorch toolkit. As part of this lecture, you will first read through linked official Pytorch tutorials. Then you will work through exercises on Tensors, Autograd, Neural Networks and Classifier Training/Evaluation. Some of the questions will ask you to implement small lines of code, while other questions will ask you to guess what the output of operations will be, or identify issues with the code. These exercises can be a great way of solidifying your knowledge of a toolkit, and I strongly encourage you to try the problems yourselves before you look them up in the referenced solutions.




Midjourney Generation: “a robot steps forward walking down a hill”

Learning outcomes:

- Perform Tensor operations in PyTorch.
- Understand the backward and forward passes of a neural network in context of Autograd.
- Detect common issues in PyTorch training code.

PyTorch Exercises

As part of this lecture, in each of the sections, you will first read through the linked official Pytorch Blitz tutorial pages. Then you will work through exercises. We will cover Tensors, Autograd, Neural Networks and Classifiers.

There are 55 exercises in total. The exercises have solutions hidden through a black highlight –  You can reveal the solution by highlighting it. You can also make a copy of the document and remove the highlights all at once. If you have any suggestions for improvement on any of the questions, you can send an email to the instructor.

Installation

First, you'll need to make sure you have all of the packages installed. Here's my environment setup:

```
conda create --name lec6 python=3.9
conda activate lec6
# MPS acceleration is available on MacOS 12.3+
conda install pytorch torchvision torchaudio -c pytorch-nightly
conda install -c conda-forge matplotlib
conda install -n lec6 ipykernel --update-deps --force-reinstall
```

Tensors

We'll start with the very basics, Tensors. First, go through the Tensor tutorial [here](#). An excerpt:

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other specialized hardware to accelerate computing. If you're familiar with ndarrays, you'll be right at home with the Tensor API. If not, follow along in this quick API walkthrough.

Think you know Tensors well? I'd like you to then attempt the following exercise. Create a notebook to solve the following exercise:

1. Create an tensor from the nested list `[[5,3], [0,9]]`

```
_____
_____
```

2. Create a tensor 't' of shape (5, 4) with random numbers from a uniform distribution on the interval [0, 1)

```
_____
```

3. Find out which device the tensor 't' is on and what its datatype is.

```
_____
_____
```

4. Create two random tensors of shape (4,4) and (4,4) called 'u' and 'v' respectively. Join them to make a tensor of shape (8, 4).

```
_____
_____
_____
```

5. Join u and v to create a tensor of shape (2, 4, 4).

```
_____
```

6. Join u and v to make a tensor, called w of shape (4, 4, 2).

```
_____
_____
```

7. Index w at 3, 3, 0. Call that element 'e'.

```
_____
```

8. Which of u or v would you find w in? Verify.

```
_____
_____
```

9. Create a tensor 'a' of ones with shape (4, 3). Perform element wise multiplication of 'a' with itself.

```

a = torch.ones(4, 3)
a = a * a

```

10. Add an extra dimension to 'a' (a new 0th dimension).

```

a = torch.unsqueeze(a, 0)

```

11. Perform a matrix multiplication of a with a transposed.

```

a = torch.matmul(a, a.T)

```

12. What would a.mul(a) result in?

```

a.mul(a)

```

13. What would a.matmul(a.T) result in?

```

a.matmul(a.T)

```

14. What would a.mul(a.T) result in?

```

a.mul(a.T)

```

15. Guess what the following will print. Verify

```

t = torch.ones(5)
n = t.numpy()
n[0] = 2
print(t)

```

```

2. 1. 1. 1. 1.

```

16. What will the following print?

```

t = torch.tensor([2., 1., 1., 1., 1.])
t.add(2)
t.add_(1)
print(n)

```

Autograd and Neural Networks

Next, we go through the [Autograd](#) tutorial and the [Neural Networks](#) tutorial. A few relevant (slightly modified) excerpts:

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by parameters (consisting of weights and biases), which in PyTorch are stored in tensors. Neural networks can be constructed using the `torch.nn` package.

Training a NN happens in two steps:

- *Forward Propagation: In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.*
- *Backward Propagation: In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (gradients), and optimizing the parameters using gradient descent.*

More generally, a typical training procedure for a neural network is as follows:

- *Define the neural network that has some learnable parameters (or weights)*
- *Iterate over a dataset of inputs*
- *Process input through the network*
- *Compute the loss (how far is the output from being correct)*
- *Propagate gradients back into the network's parameters*
- *Update the weights of the network, typically using a simple update rule: $\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$*

Equipped with these tutorials, we are ready to attempt the following exercise! Assume we have the following starter code:

```
import torch
from torchvision.models import resnet18, ResNet18_Weights
model = resnet18(weights=ResNet18_Weights.DEFAULT)
data = torch.rand(1, 3, 64, 64)
labels = torch.rand(1, 1000)
```

Create a notebook to solve the following exercises:

17. Run a forward pass through the model with the data and save it as preds.

```
[REDACTED]
```

18. What should the shape of preds be? Verify your guess.

```
[REDACTED]
```

```
[REDACTED]
```

19. Save the weight parameter of the conv1 attribute of resnet18 as 'w'. Print w because we will need it for later

```
[REDACTED]
```

```
[REDACTED]
```

```
[REDACTED]
```

20. What should the 'grad' attribute for w be? Verify.

```
[REDACTED]
```

```
[REDACTED]
```

21. Create a CrossEntropy loss object, and use it to compute a loss using 'labels' and 'preds', saved as 'loss'. Print loss because we will need it for later.

```
[REDACTED]
```

```
[REDACTED]
```

```
[REDACTED]
```

22. Print the last mathematical operation that created 'loss'.

```
[REDACTED]
```

23. Perform the backward pass.

```
[REDACTED]
```

24. Should 'w' have changed? Check with output of #3

```
[REDACTED]
```

25. Will the 'grad' attribute for w be different than #4? Verify.

```
[REDACTED]
```

```
[REDACTED]
```

26. What should 'grad' attribute for loss return for you? Verify.

```
[REDACTED]
```

27. What should the requires_grad attribute for loss be? Verify.

```
[REDACTED]
```

28. What should requires_grad for labels be? Verify.

```
[REDACTED]
```

29. What will happen if you perform the backward pass again?

```
[REDACTED]
```

30. Create an SGD optimizer object with lr=1e-2 and momentum=0.9. Run a step.

```
[REDACTED]
```

31. Should 'w' have changed? Check with output of #3

```
[REDACTED]
```

32. Should 'loss' have changed? Check with output of #5

```
[REDACTED]
```

33. Zero the gradients for all trainable parameters.

```
[REDACTED]
```

34. What should the 'grad' attribute for w be? Verify.

```
[REDACTED]
```

35. Determine, without running, whether the following code will successfully execute.

```
data1 = torch.zeros(1, 3, 64, 64)
data2 = torch.ones(1, 3, 64, 64)

predictions1 = model(data1)
predictions2 = model(data2)
```

```
l = torch.nn.CrossEntropyLoss()
loss1 = l(predictions1, labels)
loss2 = l(predictions2, labels)

loss1.backward()
loss2.backward()
```

36. As above, determine whether the following code will successfully execute.

```
data1 = torch.zeros(1, 3, 64, 64)
data2 = torch.ones(1, 3, 64, 64)

predictions1 = model(data1)
predictions2 = model(data1)

l = torch.nn.CrossEntropyLoss()
loss1 = l(predictions1, labels)
loss2 = l(predictions2, labels)

loss1.backward()
loss2.backward()
```

37. As above, determine whether the following code will successfully execute.

```
data1 = torch.zeros(1, 3, 64, 64)
data2 = torch.ones(1, 3, 64, 64)

predictions1 = model(data1)
predictions2 = model(data2)

l = torch.nn.CrossEntropyLoss()
loss1 = l(predictions1, labels)
loss2 = l(predictions1, labels)

loss1.backward()
loss2.backward()
```


38. For one(s) that don't execute, how might you modify one of the `.backward` lines to make it work?

39. What will the output of the following code?

```
predictions1 = model(data)
l = torch.nn.CrossEntropyLoss()
loss1 = l(predictions1, labels)
loss1.backward(retain_graph=True)

w = model.conv1.weight.grad[0][0][0][0]
a = w.item()

loss1.backward()
b = w.item()

model.zero_grad()
c = w.item()

print(b//a,c)
```

40. What will be the output of the following code?

```
predictions1 = model(data)
l = torch.nn.CrossEntropyLoss()
loss1 = l(predictions1, labels)
loss1.backward(retain_graph=True)

a = model.conv1.weight.grad[0][0][0][0]

loss1.backward()
b = model.conv1.weight.grad[0][0][0][0]

model.zero_grad()
c = model.conv1.weight.grad[0][0][0][0]

print(b//a,c)
```

41. What is wrong with the following code?

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub(f.grad.data * learning_rate)
```

42. Order the following steps of the training loop correctly (there are multiple correct answers, but one typical setup that you would have seen in the tutorial):

```
optimizer.step(), optimizer.zero_grad(), loss.backward(), output =
net(input), loss = criterion(output, target)
```

There are multiple correct solutions, including:

```
optimizer.zero_grad(), loss.backward(), output = net(input),
optimizer.step(), loss = criterion(output, target)
optimizer.step(), output = net(input), loss = criterion(output,
target), optimizer.zero_grad(), loss.backward()
```

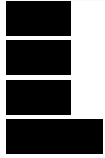
43. What will be the output of the following code?

```
net = resnet18(weights=ResNet18_Weights.DEFAULT)
data = torch.rand(1, 3, 64, 64)
target = torch.rand(1, 1000)
optimizer = torch.optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
criterion = torch.nn.CrossEntropyLoss()
orig = net.conv1.weight.clone()[0, 0, 0, 0]
weight = net.conv1.weight[0, 0, 0, 0]
# 1
optimizer.zero_grad()
print(f"{weight == orig}")

# 2
output = net(data)
loss = criterion(output, target)
print(f"{weight == orig}")

# 3
loss.backward()
print(f"{weight == orig}")
```

```
# 4
optimizer.step()
print(f"{weight == orig}")
```



44. We're going to implement a neural network with one hidden layer. This network will take in a grayscale image input of 32x32, flatten it, run it through an affine transformation with 100 out_features, apply a relu non-linearity, and then map onto the target classes (10). Implement the initialization and the forward pass completing the following piece of code. Use nn.Linear, F.relu, torch.flatten

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # your code here

    def forward(self, x):
        # your code here
        return x
```

```
[REDACTED]
```

```
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
```

```
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
```

45. Using two lines of code, verify that you're able to do a forward pass through the above network.

```
[REDACTED]
[REDACTED]
```

46. Without running the code, guess what would the following statement yield?

```
net = Net()
print(len(list(net.parameters())))
```

```
[REDACTED]
```

47. Get the names of the net parameters

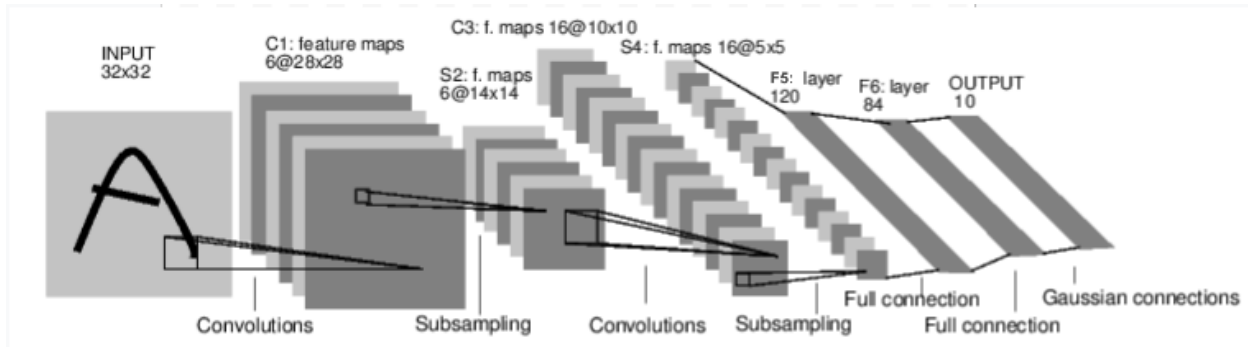
```
[REDACTED]
[REDACTED]
```

48. What network layer is the following statement referring to? What will it evaluate to?

```
print(list(net.parameters())[1].size())
```

```
[REDACTED]
```

49. The following schematic has all of the information you need to implement a neural network. Implement the initialization and the forward pass completing the following pieces of code. Use `nn.Conv2d`, `nn.Linear`, `F.max_pool2d`, `F.relu`, `torch.flatten`. Hint: the ReLUs are applied after the subsampling operations and after the first two fully connected layers.



```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # your code here

    def forward(self, x):
        # your code here
        return x
```



```

[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

```

51. Try increasing the width of your network by increasing the number of output channels of the first convolution from 6 to 12. What else do you need to change?

```

[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

```

Classifier Training

Next, we go to the final tutorial in the Blitz: the [Cifar10](#) tutorial. This tutorial trains an image classifier going through the following steps in order:

- *Load and normalize the CIFAR10 training and test datasets using torchvision*
- *Define a Convolutional Neural Network*
- *Define a loss function*
- *Train the network on the training data*
- *Test the network on the test data*

Once you have gone through the tutorial above, answer the following questions:

52. The following dataset loading code runs but are there mistakes in the following code? What are the implications of the errors? What are the fixes?

```

import torch
from torchvision import datasets, transforms

```

```

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = datasets.CIFAR10(root='./data', train=False,
                             download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                           shuffle=False, num_workers=2)

testset = datasets.CIFAR10(root='./data', train=True,
                             download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                           shuffle=False, num_workers=2)

```



53. Write 2 lines of code to get random training images from the dataloader (assuming errors above are fixed).



54. The following training code runs but are there mistakes in the following code (mistakes include computational inefficiencies)? What are the implications of the errors? What are the fixes?

```

running_loss = 0.0
for i, data in enumerate(trainloader, 0):
    # get the inputs; data is a list of [inputs, labels]
    inputs, labels = data

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # print statistics

```



```

        running_loss += loss
    if i % 2000 == 1999:    # print every 2000 mini-batches
        print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
2000:.3f}')
        running_loss = 0.0
        break

```

55. The following evaluation code runs but are there mistakes in the following code (mistakes include computational inefficiencies)? What are the implications of the errors? What are the fixes?

```

correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for
our outputs
for data in testloader:
    images, labels = data
    # calculate outputs by running images through the network
    outputs = net(images)
    # the class with the highest energy is what we choose as prediction
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum()

print(f'Accuracy of the network on the 10000 test images: {100 * correct //
total} %')

```

Cheatsheet

It may take you some time to feel comfortable with PyTorch, and that's okay! PyTorch is a powerful tool for deep learning development. After the above exercises, you can go through the Quickstart tutorial [here](#), which will cover more aspects including Save & Load Model, and

Datasets and Dataloaders. As you learn the API, you may find it useful to remember key usage patterns; a PyTorch cheatsheet I like is found [here](#).

And that covers our fundamentals of PyTorch! Congratulations – you’re now equipped to start tackling more complex deep learning code that leverages PyTorch.