Lab 1

Michael Trey Peterson

02/15/2022

Contents

1 Creating Tensors

```python
## creating a tensor of 4 rows and 3 columns consisting of ones ##
## torch.ones(row, columns) ##
print("Lab ----------------------------------------------------------------")
x = torch.ones(4, 3)
print(x)

## Create zeroes (same method as ones) ##
x = torch.zeros(4, 3)
print(x)

## User stuff ##
print("User ----------------------------------------------------------------")
a = torch.ones(1, 2) # 1s (1 row , 2 col)
b = torch.zeros(10, 5) # 0s (10 row, 5 col)
print(a)
print(b)

## Random Starting Values ##
## We often set a specific value as random seed ##
print("Lab ----------------------------------------------------------------")
torch.manual_seed(3)
## generate tensor populated by random values
x = torch.rand(4, 3)
print(x)
x = torch.randn(4,3)
print(x)
## User Stuff##
print("User ----------------------------------------------------------------")
torch.manual_seed(2)
x = torch.rand(5, 1)
print(x)
```

Figure 1: Tensor Creation Code

```
Lab ------------------------------------------------------------------
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
User ------------------------------------------------------------------
tensor([[1., 1.]])
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
Lab ------------------------------------------------------------------
tensor([[0.0043, 0.1056, 0.2858],
        [0.0270, 0.4716, 0.0601],
        [0.7719, 0.7437, 0.5944],
        [0.8879, 0.4510, 0.7995]])
tensor([[ 0.3375,  1.0111, -1.4352],
        [ 0.9774,  0.5220,  1.2379],
        [-0.8646,  0.2990,  0.4192],
        [-0.0799,  0.9264,  0.8157]])
User ------------------------------------------------------------------
tensor([[0.6147],
        [0.3810],
        [0.6371],
        [0.4745],
        [0.7136]])
```

Figure 2: Tensor Creation Output

2 Tensor Operations

Tensor Operation Math and in-place operation

```
print("Lab ----------------------------------------------------------------")
#Create two tensors
x = torch.ones([3, 2])
y = torch.ones([3, 2])
#Adding two tensors
z = x + y
z = torch.add(x,y)
print(z)
#subtract two tensors
z = x - y
z = torch.sub(x, y)
print(z)
#Create two tensors
x = torch.ones([3, 2])
y = torch.ones([3, 2])
#inplace operation
z = y.add_(x)
print(z)

print("User ----------------------------------------------------------------")
torch.manual_seed(1)
x = torch.rand(3, 2)
y = torch.rand(3, 2)
z = torch.add(x, y)
print(z)
z = torch.sub(x, y)
print(z)
```

Figure 3: Tensor Operation Code

```
Lab ----------------------------------------------------------------
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
User ----------------------------------------------------------------
tensor([[1.1548, 1.0337],
        [0.9726, 1.1735],
        [0.6680, 1.3245]])
tensor([[ 0.3605, -0.4751],
        [-0.1664,  0.2959],
        [-0.6094,  0.2752]])
```

Figure 4: Tensor Operation Output

Tensor Operation Reshaping

```python
print("Lab -----------------------------------------------------------------")
## create tensor with 3 row and 2 column
x = torch.tensor([[1,2], [3,4], [5,6]])
## reshaping to 2 rows and 3 columns
y = x.view(2, 3)
print(y)
## use of -1 to reshape a tensor
y = x.view(6, -1)
print(y)
print("User -----------------------------------------------------------------")
y = x.view(3, 2)
print(y)
y = x.view(-1, 6)
print(y)
```

```
Lab -----------------------------------------------------------------
tensor([[1, 2, 3],
        [4, 5, 6]])
tensor([[1],
        [2],
        [3],
        [4],
        [5],
        [6]])
User -----------------------------------------------------------------
tensor([[1, 2],
        [3, 4],
        [5, 6]])
tensor([[1, 2, 3, 4, 5, 6]])
```

Figure 5: Tensor Reshaping

Tensor Operation Slicing

```python
## create a tensor with certain values
print("Lab -----------------------------------------------------------------")
x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
## slice and print slice ##
print(x[:, 2]) # print every row and last column of x
print(x[0, :]) # Every Column and first row
y = x[:, 1] # take the elements in the second columns and create another tensor
print(y)
## user slicing and print slice ##
## create a tensor with certain values
print("User -----------------------------------------------------------------")
print(x[2, 1])
print(x[:, 0])
y = x[2, 2]
print(y)
```

```
Lab -----------------------------------------------------------------
tensor([3, 6, 9])
tensor([1, 2, 3])
tensor([2, 5, 8])
User -----------------------------------------------------------------
tensor(8)
tensor([1, 4, 7])
tensor(9)
```

Figure 6: Tensor Slicing

3 CUDA + autograd

CUDA Operations

```
[ ] ## check the number of CUDA supported GPU that are connected to the machine
    print(torch.cuda.device_count())
    ## get the name of the GPU Card
    print(torch.cuda.get_device_name(0))
    ## assign cuda GPU located at location '0' to a variable
    cuda0 = torch.device('cuda:0')
    ##performing operations on GPU
    a = torch.ones(3, 2, device=cuda0)
    b = torch.ones(3, 2, device=cuda0)
    c = a + b
    print(c)
    ## move the result to CPU
    c = c.cpu()
    print(c)
    ### Operation ###
    ### CPU -> parameters -> GPU -> operation -> result -> CPU
```

```
1
Tesla T4
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]], device='cuda:0')
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
```

Figure 7: CUDA Code

Differentiation and Gradience

```
## create tensor with requires
## This will track all the operations
x = torch.ones([3,3], requires_grad = True)
print(x)

## perform a tensor addition and check the result
y = x + 1
print(y)

## perform a tensor multiplaction and check the result
z = y*y + 1
print(z)

## Adding all the values in Z and check the result
t = torch.sum(z)
print(t)

## perform backpropation (partial derivate of t with respect to x) and check the result
t.backward()
print(x.grad)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], requires_grad=True)
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]], grad_fn=<AddBackward0>)
tensor([[5., 5., 5.],
        [5., 5., 5.],
        [5., 5., 5.]], grad_fn=<AddBackward0>)
tensor(45., grad_fn=<SumBackward0>)
tensor([[4., 4., 4.],
        [4., 4., 4.],
        [4., 4., 4.]])
```

Figure 8: Gradience code

4 Toy Example

The following shows code made to train a linear model in 4 iterations. The linear model consists of the toy example presented in the lab as a way to make sure the model is functioning correctly (results are cross-referenced with lecture). The model takes an input and weight tensor of size n and then performs a dot product, the product is alias'd as z and the sum is alias'd as t. Since the model is linear an activation function isn't needed. To perform training the error is found by taking the target - result, error, and then multiplying the error by the learning rate and input (with it's corresponding weight) to give a delta weight. The delta weight is then added to the original weight.

Toy Example linear neuron

```python
iter = 4
# inputs
input = torch.tensor([[5.0,2.0,4.0], [3.0, 3.0, 3.0], [0.0, 5.0, 1.0], [2.0, 1.0, 2.0]])
target = torch.tensor([[1250], [900], [350], [550]])
learn_rate = [1/70, 1/12, 1/27, 2/20]
# initial weights
w = torch.tensor([50.0, 50.0, 50.0], requires_grad=True)
err_arr = []
# algorith
# z = (i * w)
# t = sum(z)
for i in range(0, iter):
  print("Iteration: ", i)
  print(input[i, :])
  # perform weighted multiplication [i * w]
  z = input[i,:] * w
  t = torch.sum(z)
  print(t)
  print("sum: ", t)
  print("target: ", target[i])
  # target - prediction
  err = torch.sub(target[i], t)
  err_arr.append(err)
  print("error: ", err)
  # apply backpropagtion
  #t.backward()
  #print(w.grad)
  w = w + (err * input[i,:] * learn_rate[i])
  print("\n\n")

plt.plot(err_arr, "-*")
plt.title("Error vs Iterations")
```

Figure 9: Linear Neuron Code

As shown the results correspond with the toy example slides to demonstrate successful training of the linear model.

```
Iteration:  0
tensor([5., 2., 4.])
tensor(550., grad_fn=<SumBackward0>)
sum:  tensor(550., grad_fn=<SumBackward0>)
target:  tensor([1250])
error:  tensor([700.], grad_fn=<SubBackward0>)



Iteration:  1
tensor([3., 3., 3.])
tensor(780., grad_fn=<SumBackward0>)
sum:  tensor(780., grad_fn=<SumBackward0>)
target:  tensor([900])
error:  tensor([120.], grad_fn=<SubBackward0>)



Iteration:  2
tensor([0., 5., 1.])
tensor(620., grad_fn=<SumBackward0>)
sum:  tensor(620., grad_fn=<SumBackward0>)
target:  tensor([350])
error:  tensor([-270.], grad_fn=<SubBackward0>)



Iteration:  3
tensor([2., 1., 2.])
tensor(530., grad_fn=<SumBackward0>)
sum:  tensor(530., grad_fn=<SumBackward0>)
target:  tensor([550])
error:  tensor([20.], grad_fn=<SubBackward0>)
```

Figure 10: Gradience Result

A visual representation of the errorr shows convergence to 0 meaning that the model is training and the input/target have a realizable pattern.

```
Text(0.5, 1.0, 'Error vs Iterations')
```

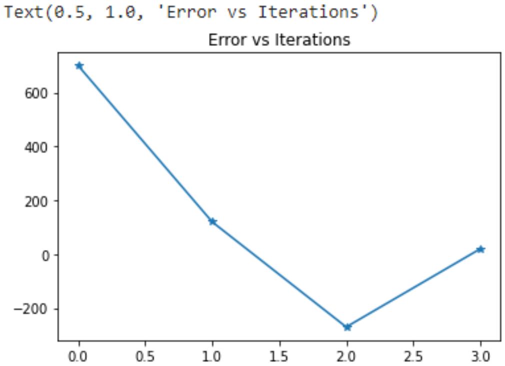

Figure 11: Error Graph