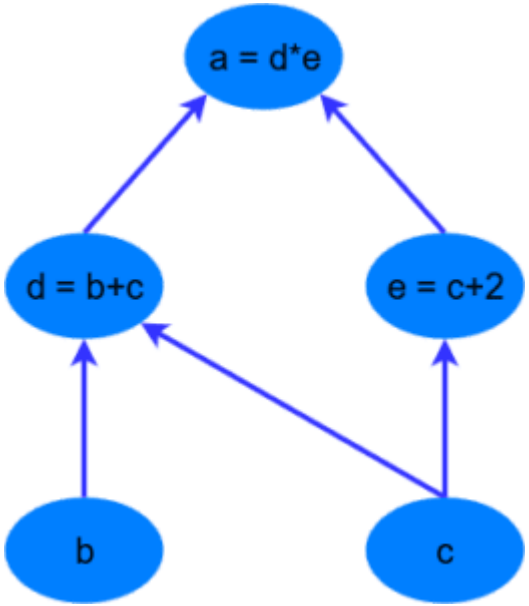
Введение в PyTorch

Дмитрий Буряк

к.ф.-м.н.

dyb04@yandex.ru

Динамический граф вычислений



*a = (b + c) * (c + 2)*

- ❑ Выражение разбивается на части, которые вычисляются последовательно.
- ❑ Каждый узел графа: независимый кусок кода, которому для работы нужны лишь входы.
- ❑ Преимущества: параллельная обработка.
- ❑ Технически: тяжелые вычисления (свертки, умножения матриц) реализованы на C/C++ / CUDA C++.

Тензоры

❑ Основная вычислительная единица - тензор, можно представить как вложение произвольного количества динамических массивов переменной размерности.

❑ Инициализация: 3 матрицы размера 2x2

```
In [1]: import torch
In [2]: x=torch.Tensor(3,2,2)
In [3]: print(x)
tensor([[[8.7725e+29, 3.0904e-41],
         [0.0000e+00, 0.0000e+00]],

        [[       nan, 0.0000e+00],
         [1.3733e-14, 6.4069e+02]],

        [[4.3066e+21, 1.1824e+22],
         [4.3066e+21, 6.3828e+28]]])
```

❑ Матричное умножение

```
In [18]: a=torch.rand(2,3)
In [19]: b=torch.rand(3,2)
In [20]: c=torch.mm(b,a)
In [21]: print(c)
Out[21]:
tensor([[0.4033, 0.0286, 0.1627],
        [0.6677, 0.0475, 0.2699],
        [0.5106, 0.0588, 0.3515]])
```

```
In [38]: a=a*0+1
In [39]: b=b*0+2
In [40]: 3+torch.mm(b,a)
Out[40]:
tensor([[7., 7., 7.],
        [7., 7., 7.],
        [7., 7., 7.]])
```

Тензоры и вычислительные устройства

- 2 типа: на CPU и на GPU

```
In [73]: a=torch.rand(3,3)
In [74]: a
Out[74]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]])
In [75]: a=a.cuda(device = 0)
In [76]: a
Out[76]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]],
device='cuda:0')
In [77]: a=a.cpu()
In [78]: a
Out[78]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]])
```

```
In [79]: a=torch.rand(3,3)

In [80]: b=torch.rand(3,3).cuda(0)

In [81]: a+b
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-81-ca730b97bf8a> in <module>
----> 1 a+b
RuntimeError: expected type torch.FloatTensor but got
torch.cuda.FloatTensor
```

- Операции между тензорами должны выполняться на одном устройстве

  torch.cuda.is_available()

  torch.cuda.device_count()

  torch.cuda.set_device(device)

Автоматическое дифференцирование

```
In [42]: from torch.autograd import Variable
In [43]: x=Variable(torch.ones(2,2)*2,requires_grad = True)
In [44]: z = 2 * (x * x) + 5 * x
In [45]: z.backward(torch.ones(2, 2))
In [46]: print(x.grad)
tensor([[13., 13.],
        [13., 13.]])
In [47]: z
Out[47]:
tensor([[18., 18.],
        [18., 18.]], grad_fn=<AddBackward0>)
```

dz/dx = 4x+5,
При x_i,j =2 dz/dx должен быть заполнен числами 13

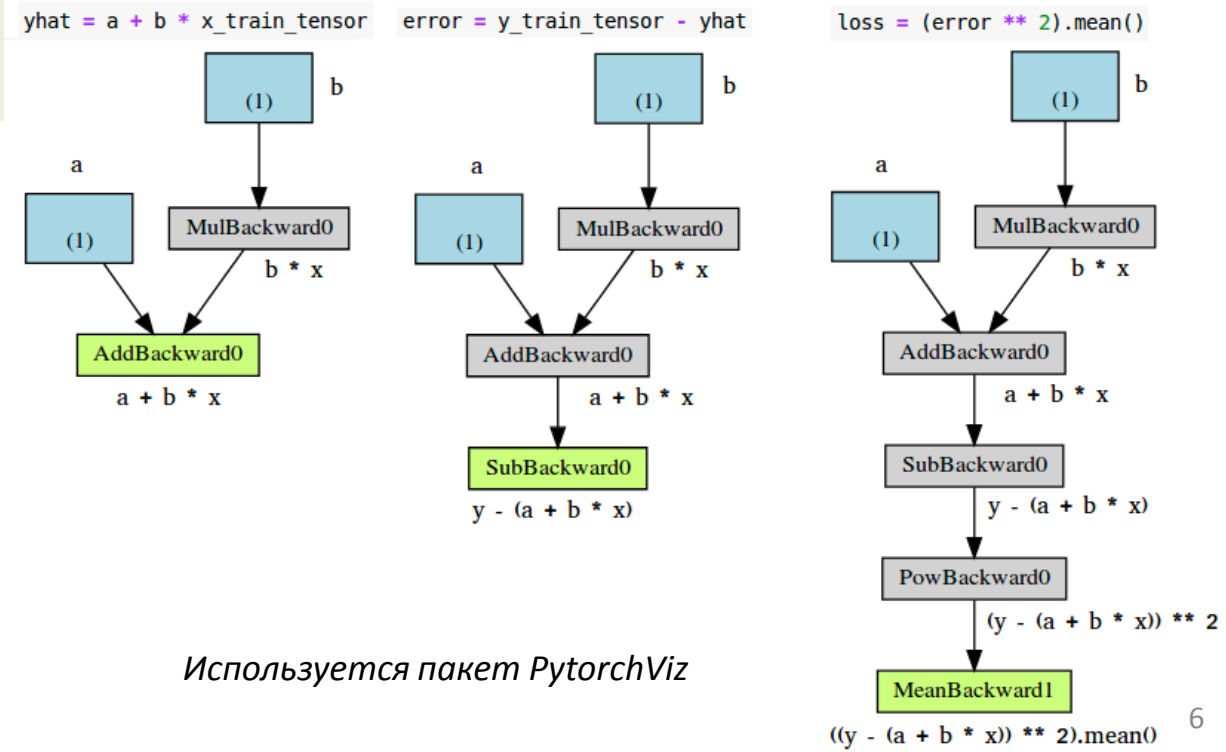
```
In [62]: x.grad
Out[62]:
tensor([[13., 13.],
        [13., 13.]])
In [63]: x.grad_fn
In [64]: z.grad
In [65]: z.grad_fn
Out[65]:
<AddBackward0 at
0x7f681bbbaa20>
```

Компонент тензора .grad содержит градиент:
значение или функцию

- Как это реализовано: все математические операции (+,-,*,/…) с тензорами перегружены.
- Каждая операция с тензорами проверяет, нужно ли вычислять градиент (компонент класса torch.Tensor requires_grad == True)
- Каждый torch.Tensor содержит компоненты:
  grad (значение градиента) или
  grad_fn(ссылка на функцию вычисления градиента)
- Вызов z.backward() запускает проход по графу вычислений в обратную сторону, с вызовом функций grad_fn

Примеры вычислительных графов в представлении Pytorch (1)

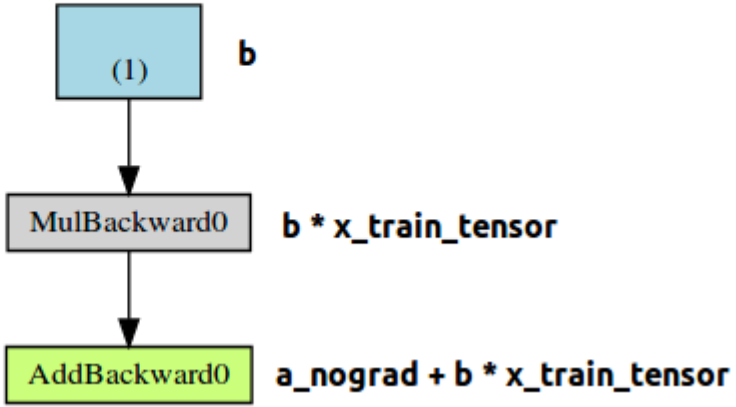
```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
 b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
yhat = a + b * x_train_tensor
error = y_train_tensor - yhat
loss = (error ** 2).mean()
```



Используется пакет PytorchViz

Примеры вычислительных графов в представлении Pytorch (2)

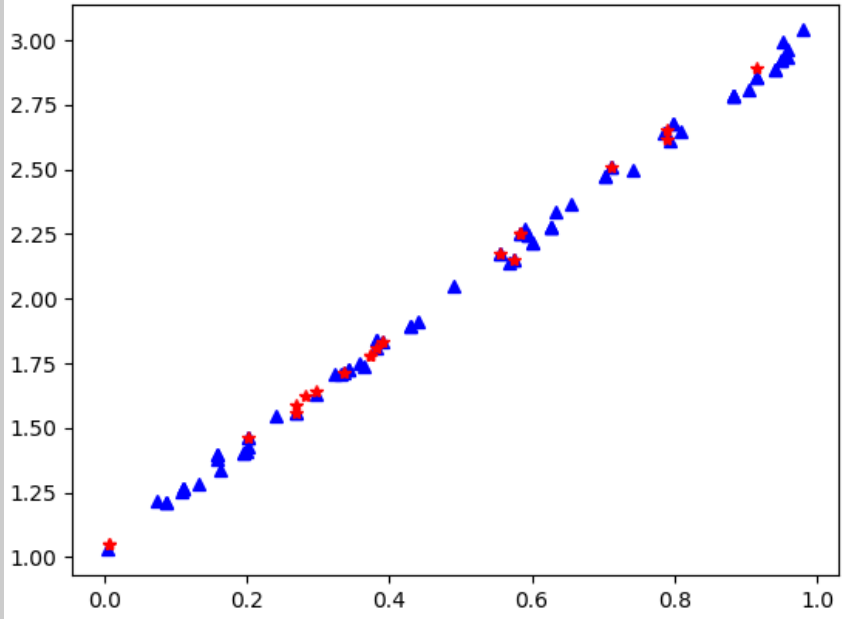
```
a_nograd = torch.randn(1, requires_grad=False, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

yhat = a_nograd + b * x_train_tensor
```



Простейшая оптимизация параметров: линейная регрессия (1)

Создаем набор данных:

```python
import numpy
import torch
from matplotlib import pyplot as plt
# Data Generation
torch.manual_seed(42)

x = torch.rand(100, 1)
y = 1 + 2 * x + 0.1 * torch.rand(100, 1)

# Shuffles the indices
idx = (torch.rand(100)*100).long()


# Uses first 80 random indices for train
train_idx = idx[:80]
# Uses the remaining indices for validation
val_idx = idx[80:]

# Generates train and validation sets
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]

plt.figure("train")
plt.plot(x_train.numpy(),y_train.numpy(),"b",lw=0,marker="^")
plt.plot(x_val.numpy(),y_val.numpy(),"r",lw=0,marker="*")
plt.show()
```



Простейшая оптимизация параметров: линейная регрессия (2)

```
31 torch.manual_seed(42)
32
33 a = torch.randn(1, requires_grad=True, dtype=torch.float)
34 b = torch.randn(1, requires_grad=True, dtype=torch.float)
35 print("Initial values:",a, b)
36
37 lr = 1e-1
38 n_epochs = 1000
39
40 # Defines a SGD optimizer to update the parameters
41 optimizer = optim.SGD([a, b], lr=lr)
42
43 for epoch in range(n_epochs):
44     yhat = a + b * x_train
45     error = y_train - yhat
46     loss = (error ** 2).mean()
47
48     loss.backward()
49
50     # Manual variant: gradient step
51     # with torch.no_grad():
52     #     a -= lr * a.grad
53     #     b -= lr * b.grad
54     optimizer.step()
55
56     # Manual variant: cleaning gradients
57     # a.grad.zero_()
58     # b.grad.zero_()
59     optimizer.zero_grad()
60
61 print("optimized values:",a,b)
62 with torch.no_grad():
63     print("error:",torch.mean((y_train - (a+b*x_train))**2))
64     print("validation error:",torch.mean((y_val- (a+b*x_val))**2))
```

Пакет torch.optim -- коллекция различных оптимизаторов градиентного спуска, от простейшего SGD до современных:

- optim.ASGD,
- optim.Adadelta,
- optim.Adagrad,
- optim.RMSprop,
- optim.Adam

```
Initial values: tensor([0.3367], requires_grad=True) tensor([0.1288], requires_grad=True)
optimized values: tensor([1.0422], requires_grad=True) tensor([1.9991], requires_grad=True)
error: tensor(0.0007)
validation error: tensor(0.0005)
```

torch.nn: коллекция модулей-компонентов нейронных сетей

- nn.Sequential – коллекция модулей
- nn.Linear – реализация перцептрона
- nn.Parameter – тензор-параметр, приспособленный для работы с torch.nn
- nn.Conv2d: 2-мерная свертка
- nn.Dropout: дропаут
- nn.ReLU,nn.PReLU,nn.Tanh,nn.Sigmoid, nn.Softmax – функции активации

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

class Model(nn.Module):
    def __init__(self,size):
        super(Model, self).__init__()
        self.mlp1 = nn.Linear(size, size)
        self.mlp2 = nn.Linear(size,1)

    def forward(self, x):
        x = F.relu(self.mlp1(x))
        return F.relu(self.mlp2(x))
```

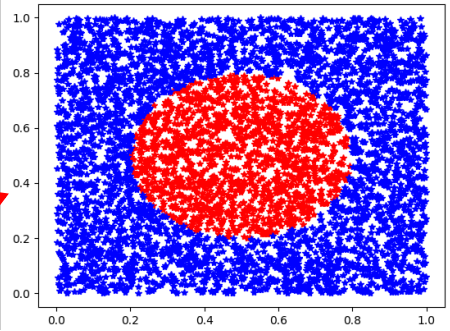
mlp_net1.py

```
In [1]: import torch

In [2]: from mlp_net1 import Model

In [3]: a=torch.rand(3)

In [4]: m=Model(3)
In [5]: b=m(a)
In[6]: b.size()
Out[7]: torch.Size([1])
```

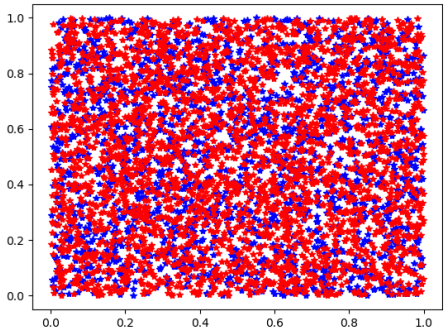
```
In [9]: m.mlp1
Out[9]: Linear(in_features=3,
out_features=3, bias=True)
In [10]: m.mlp1.weight
Out[10]:
Parameter containing:
tensor([[-0.5699, -0.0745,  0.3962],
        [-0.0680,  0.4040,  0.3050],
        [-0.3282,  0.0891, -0.2783]],
requires_grad=True)
```

Классификация данных при помощи MLP (1)

```python
import numpy
import torch
import torch.optim as optim
from matplotlib import pyplot as plt
# Data Generation
torch.manual_seed(42)


x = torch.rand(5000)
y = torch.rand(5000)

center = (0.5,0.5)
radius = 0.3

distance = torch.sqrt( (center[0]-x)**2+(center[1]-y)**2 )
pos_labels = distance <= radius
neg_labels = distance > radius

labels = torch.zeros(5000)

labels[pos_labels] = 1


pos_data_x,pos_data_y = x[pos_labels],y[pos_labels]
neg_data_x,neg_data_y = x[neg_labels],y[neg_labels]

plt.figure("All data")
plt.plot(pos_data_x.numpy(),pos_data_y.numpy(),"r",marker="*",lw=0)
plt.plot(neg_data_x.numpy(),neg_data_y.numpy(),"b",marker="*",lw=0)
```




```python
N_train = 3000

train_data_x = x[:N_train]
train_data_y = y[:N_train]


test_data_x = x[N_train:]
test_data_y = y[N_train:]

plt.figure("Train and test data")
plt.plot(test_data_x.numpy(),test_data_y.numpy(),"b",marker="*",lw=0)
plt.plot(train_data_x.numpy(),train_data_y.numpy(),"r",marker="*",lw=0)
plt.show()
```



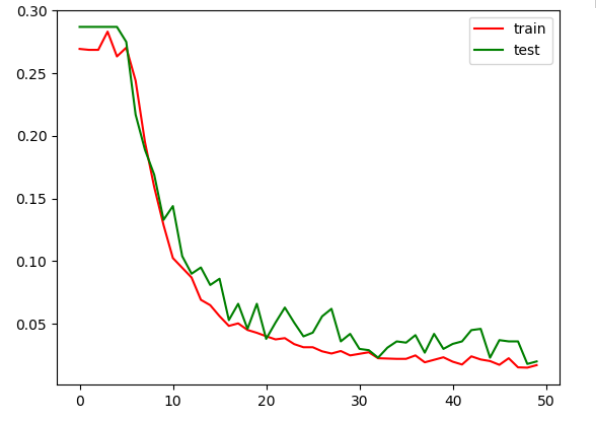
Классификация данных при помощи MLP (2)

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

class MLP(nn.Module):
    def __init__(self,size,out_size):
        super(MLP, self).__init__()
        self.mlp1 = nn.Linear(size, 15)
        self.mlp2 = nn.Linear(15,out_size)
        self.func = nn.ReLU()

    def forward(self, x):
        x = self.func(self.mlp1(x))
        return self.mlp2(x)

def output_to_label(output,softmax):

    with torch.no_grad():
        output2=softmax(output.detach())
        rez=torch.sort(output2.cpu(),1,True)
    ret_labels = []
    for i in range(rez[1].size(0)):
        ind=rez[1].data[i][0]
        lab=int(ind)
        ret_labels+=[lab]
    return ret_labels
```

mlp_net1.py

Классификация данных при помощи MLP (3)

```
from mlp_net1 import MLP
from mlp_net1 import output_to_label
import torch.nn as nn
from torch.autograd import Variable

model = MLP(2,2)

lr = 0.001
# Defines a optimizer to update the parameters
optimizer = optim.Adam(model.parameters(), lr=lr)
#combines logsoftmax and negative log likelihood loss
model_loss = nn.CrossEntropyLoss()
softmax = nn.Softmax(-1)

batch_size = 15
n_iterations = train_data_all.size(0) // batch_size
print("n iter:",n_iterations)

train_errors = []
test_errors = []
```




```
for epoch in range(50):

    print("### epoch:",epoch)
    epoch_loss = 0

    error = 0
    for iter in range(n_iterations):

        batch_idx = (torch.rand(batch_size)*train_data_all.size(0)).long()
        batch = Variable(train_data_all[batch_idx].contiguous(),requires_grad = True)
        batch.requires_grad = True


        ref = train_labels[batch_idx]

        out = model(batch)

        loss = model_loss(out,ref)

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

        lab = torch.Tensor(output_to_label(out,softmax)).long()
        error+= torch.sum(torch.abs(ref-lab))

        epoch_loss+=loss.item()

    train_errors+=[ [epoch,error.item()/(n_iterations*batch_size)] ]
    print(error,(n_iterations*batch_size))
    #tests
    if (epoch%1 ==0):
        test_error = 0
        with torch.no_grad():
            batch = test_data_all
            out = model(batch)
            lab = torch.Tensor(output_to_label(out,softmax)).long()
            test_error= torch.sum(torch.abs(test_labels.long()-lab))
            test_errors+=[ [epoch,test_error.item()/test_labels.size(0)] ]
            print("test error:",test_error," of ",test_labels.size(0))
```