Fall 2018

Jupyter: Intro to Data Science - Lecture 13 DL Lab

Grant Long
CUNY City College

NYC Tech-in-Residence Corps

- The empty ipynb for you to start from in Grant's repo: https://github.com/grantmlong/itds2018 (https://github.com/grantmlong/itds2018)

Setting the stage

In [ ]:
```
import sys
sys.version
```

In [ ]:
```
# if running on colab, we'll install pytorch right here right now
# if running locally, pip install this in your conda environment.
# note we're installing the bleeding edge 1.0.0 pre-release, official relea
# !pip install torchvision
# !pip install torch_nightly -f https://download.pytorch.org/whl/nightly/cu
```

In [ ]:
```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.utils.data import TensorDataset, DataLoader
import torch.nn.functional as F
print(torch.__version__)

%matplotlib inline

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
```

Torch and autograd basics

Torch is a package that defines vectors, matrices, or in general "tensors". If you know numpy, you will not be surprised by any of these:

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

In [ ]:
```
b = torch.arange(9).float().view(3,3)
b
```

In [ ]:
```
(a+b)**2
```

In [ ]:
```
b[:,0]
```

In [ ]:
```
a.zero_() # operations with an underscore modify the Tensor in place.
a
```

You can slice and dice tensors and they have roughly all tensor operations you expect equivalently to numpy, but with a bit more low level control. If you need more intro: https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py)

So what's the big deal about pytorch?

autograd = automatic differentiation.

Every `torch.Tensor` , let's say `x` , has an important flag `requires_grad` . If this flag is set to True, pytorch will keep track of the graph of operations that happen with this tensor. When we finally arrive at some output (a scalar variable based on a sequence of operations on `x`), we can call `.backward()` on this output, to compute the gradient `d(output) / dx` . This gradient will end up in `x.grad` .

In [ ]:
```
x = torch.randn(2,2, requires_grad=True)
x
```

In [ ]:
```
y=(x**2 + x)
z = y.sum()
z
```

We know from high school math that the derivative `dz / dx[i,j]` = 2*x +1

In [ ]:
```
z.backward()
x.grad
```

In [ ]:
```
2*x+1
```

What about the intermediate variable y? Does it require a gradient?

In [ ]:
```
y.requires_grad
```

However the gradient of y is not exposed, since it is an intermediary variable, the result of an operation on leaf variables. Leaf variables are inputs to the operations: the data X or the `Parameter` s of a neural network.

More about autograd in the tutorial https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py) and the docs https://pytorch.org/docs/stable/autograd.html (https://pytorch.org/docs/stable/autograd.html)

In the lecture we talked about how derivatives and backpropagation (based on the chain rule of

differentiation) play a central role in deep learning. You can now start to see how this autograd will be massively powerful to define neural networks with weights `W` and optimize them based on the gradients in `W.grad`.

Let's try this for a simple linear mapping `y = W x`, where we want to optimize W:

In [ ]:
```
torch.manual_seed(23801)
x = torch.Tensor([ [1, 0], [2, 0], [3, 0], [-3, 0], [-2, 0] ]) + 0.3*torch.
y = torch.Tensor([3, 6, 9, -9, -6])
# clearly the right relationship is y  = 3*x[:,0] + 0 * x[:,1].  So we can
W = torch.randn(1,2, requires_grad=True)
# we start W at random initialization, the gradient will point us in the ri
W
```

In [ ]:
```
ypred = x @ W.t()
print(ypred.squeeze())
loss = ((ypred.squeeze()-y)**2).mean() # mean squared error.
loss += 0.1 * W.norm() # stabilization term leading to weight decay
loss.backward()
print(W.grad)
# let's move W in that direction
W.data -= 0.1 * W.grad.data
# TODO reset gradient to zero.
print(W)
```

you can re-execute this cell above a couple of times and see how W oscillates around the optimal value of `[3,0]`.

torch defines `Module`s which do two things: (a) they contain the learnable weight, and (b) define how they operate on an input tensor to give an output. In this case this would be a `Linear` layer, reducing 2D datapoints `x` to 1D output `y`.

In [ ]:
```
linear = nn.Linear(2,1, bias=False)
linear.weight.data.copy_(W) # we re-initialize the linear layer with the W
ypred = linear(x)
ypred
```

Now you could do the same thing, compute MSE loss and backprop, then update the linear layer's weight which you can access like this:

In [ ]:
```
linear.weight
```

Revisiting KIVA logistic regression from lab 6/7

sklearn solution

In [ ]:
```
# if you downloaded the kiva data locally, you can skip wget and just set t
#!wget https://grantmlong.com/data/kiva_kenya_sample.csv
```

In [ ]:
```
# So here i copy pasted the lab 6/7 logistic regression code where we leave
# model fitting to sklearn.
data_path = 'kiva_kenya_sample.csv'
df = pd.read_csv(data_path)
print(df.shape)
print(list(df))
df['success'] = (df.STATUS=='funded')*1
df['posted_year'] = pd.to_datetime(df.POSTED_TIME).dt.year
df['posted_duration'] = (pd.to_datetime(df.PLANNED_EXPIRATION_TIME)
                             - pd.to_datetime(df.POSTED_TIME)
                            ).dt.days
model_columns = ['LOAN_AMOUNT', 'posted_year', 'posted_duration', 'LENDER_T
valids = df[model_columns].notna().all(axis=1)
X = df.loc[valids, model_columns].values
X_scaled = preprocessing.scale(X) # zero mean, unit variance

y = df.loc[valids, 'success'].values
X_train_np, X_test_np, y_train_np, y_test_np = train_test_split(
    X_scaled, y, test_size=0.2, random_state=1235)

# OK data is ready, set up the logistic regression classifier and train it.
clf = LogisticRegression(random_state=20181022, multi_class='multinomial',
clf.fit(X_train_np, y_train_np)
W, b = clf.coef_, clf.intercept_
print('The train accuracy of the sklearn model is %0.1f%%' % (clf.score(X_t
print('The test  accuracy of the sklearn model is %0.1f%%' % (clf.score(X_t
print('The learned model: y=W*x+b where \nW={} and \nb={}'.format(W,b))
```

- How many parameters (weights) does our logistic regression model have? How many datapoints did we train on?
- Old-skool machine learning rule of thumb is: you can optimize about as many parameters (weights) as you have datapoints before you can memorize the dataset (thus overfit heavily). Are we close to the limit?
- Does the model overfit?
- In deep neural networks you can easily have way more parameters than datapoints. Is overfitting an issue for neural networks?

In [ ]:
```
# number of parameters is defined by weight and bias dimensionalities.
```

torch version of logistic regression

Ok so above we used the sklearn implementation for logistic regression, which optimizes

$$\mathcal{L}(\mathcal{W}, b) = \sum_{x, y_t \in D} \ell(x, y_t; W, b)$$

with

$$\ell(x, y_t; W, b) = \mathrm{CE}(\sigma(Wx + b) || y_t)$$

So per sample, the sigmoid of the linear transformation $\sigma(Wx + b)$ is the model's prediction of the probability. This is being compared to the true label y_t, by the Cross-Entropy loss: $\text{CE}(\sigma(Wx + b)||y_t)$.

Now we'll set up a small neural network, and optimize the same loss with SGD. We will compare the solution to the one found by sklearn.

We will follow the typical training procedure for a neural network which 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: weight = weight - learning_rate * gradient

In [ ]:
```
X_train, X_test = torch.from_numpy(X_train_np).float(), torch.from_numpy(X_
y_train, y_test = torch.from_numpy(y_train_np).int(),   torch.from_numpy(y_
# we will define a "neural network" of 1 layer:
torch.manual_seed(1238)
net = nn.Linear(4,1) # computes W X + b where X is a (batch of) 4D vectors
sigmoid = nn.Sigmoid()
loss = nn.BCELoss() # Binary Cross Entropy Loss
```

Print the value of the weight and the bias of the network. Do they have any gradients right now?

In [ ]:
```
# For most modules, they're called .weight and .bias
# gradients are an attribute of a Parameter (weight) and are accessible in
```

In [ ]:
```
def print_accs(net, s=''):
    global X_train, X_test, y_train, y_test
    # net(x) is continuous value, threshold at 0 to make binary prediction
    pred_train = (net(X_train) > 0).squeeze().int()
    acc_train = (pred_train == y_train).float().mean()
    print('Train accuracy is %0.1f%%  - %s' % (acc_train*100, s))
    net.eval()
    pred_test = (net(X_test) > 0).squeeze().int()
    acc_test = (pred_test == y_test).float().mean()
    print('Test  accuracy is %0.1f%%  - %s' % (acc_test*100, s))
    net.train()
# let's print the accuracies for the untrained net, which will be random gu
print_accs(net, 'just initialized')
```

In [ ]:
```
# sklearn hides everything from you. Let's copy the weights into a torch ne
# and compute accuracies ourselves.
refnet = nn.Linear(4,1)
W, b = clf.coef_, clf.intercept_
refnet.weight.data.copy_(torch.from_numpy(W))
refnet.bias.data.copy_(torch.from_numpy(b))
print(refnet.weight) # from the sklearn optimized model
print_accs(refnet, 'sklearn weights copied over into refnet')
```

Now we'll optimize the weights and bias with SGD. Dataset and DataLoader are abstractions to help us iterate over the data in random order. We will do 1 epoch, i.e. we go through the data only once, in minibatches of size 32.

In [ ]:
```
torch.manual_seed(1238)
net = nn.Linear(4,1) # re initialize the net from scratch
print('Just initialized: weight: ', net.weight, '\nbias: ', net.bias)
w, b = net.weight, net.bias
lr = 1.0
dl = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=Tru
print_accs(net, 'before epoch')
for x,ytarget in dl:
    ypred = sigmoid(net(x).squeeze(1)) # forward, pytorch constructs the gr
    output = loss(ypred, ytarget.float()) # forward, pytorch constructs the
    output.backward() # backward, pytorch computes net.weight.grad and net.
    # access to the weight & bias tensors outside of pytorch autograd
    w, grad_w, b, grad_b = net.weight.data, net.weight.grad.data, net.bias.
    # TODO: do an SGD step for both weight and bias: w -= lr * grad
    # TODO: manually clear the gradient of the weight and the bias (use `.z
    # we need to do this before doing the forward and backward pass.
print_accs(net, 'after epoch')
print('weight: ', net.weight, '\nbias: ', net.bias)
```

Ok doing this manually gives you insight what happens down to the gradienst. But usually we do not do these things manually, it would become very cumbersome if the net becomes more complex than the simple linear layer. pytorch gives us primitives to do the same: `net.zero_grad()` to clear the gradients, and for optimization you can do `optimizer.step()` to do a step of SGD. Again we will do 1 epoch.

In [ ]:
```
torch.manual_seed(1238)
net = nn.Linear(4,1) # re initialize the net from scratch
lr = 1.0
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
dl = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=Tru
print_accs(net, 'before epoch')
for x,ytarget in dl:
    ypred = sigmoid(net(x).squeeze(1))
    output = loss(ypred, ytarget.float())
    output.backward()
    # TODO use the gradients to do a step (use the optimizer)
    # TODO clear the gradient
print_accs(net, 'after epoch')
print('weight: ', net.weight, '\nbias: ', net.bias)
```

Ok now let us redo this but for a real 3-layer neural network. You can re-execute the cell a couple of times to do more iterations and see the accuracy improve.

In [ ]:
```
torch.manual_seed(1248)
net = nn.Sequential(
    # first layer
    nn.Linear(4,32),
    nn.Dropout(0.2),
    nn.ReLU(),
    nn.Linear(32,32),
    nn.Dropout(0.2),
    nn.ReLU(),
    # output layer going to 1 prediction
    nn.Linear(32,1),
)
lr = 1.0
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
dl = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=Tru
print(net)
```

In [ ]:
```
# Do 1 epoch:
print_accs(net, 'before epoch')
for x,ytarget in dl:
    ypred = sigmoid(net(x).squeeze(1))
    output = loss(ypred, ytarget.float())
    output.backward()
    optimizer.step()
    net.zero_grad()
print_accs(net, 'after epoch')
```

And now the final real deal, we train for 10 epochs and cut the learning rate in half between epochs

In [ ]:
```
torch.manual_seed(1248)
net = nn.Sequential(
    nn.Linear(4,16),
#     nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(16,32),
#     nn.Dropout(0.1),
    nn.ReLU(),
    nn.Linear(32,16),
    nn.ReLU(),
    # output layer going to 1 prediction
    nn.Linear(16,1),
)
lr = 1.0
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0
dl = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=Tru
print_accs(net, 'Before training')
for epoch in range(10):
    scheduler.step()
    for x,ytarget in dl:
        ypred = sigmoid(net(x).squeeze(1))
        output = loss(ypred, ytarget.float())
        output.backward()
        optimizer.step()
        net.zero_grad()
    print_accs(net, 'After epoch {}'.format(epoch))
```

Voila and that's how it's done. You can ask yourself some more questions:

- What do we get back from `net.parameters()` : which trainable weights and biases does the network have now?
- How many total parameters?
- What happens if you add layers or change the ReLU activation by Sigmoid activation?
- What does the Dropout layer do?

There are many things you can play around with and where you can dig deeper.

In [ ]:
```
```

Now the real stuff: MNIST classification

MNIST is a dataset of 50k handwritten digits (0-9) which is very commonly used in the deep learning community. It is small enough to work with locally and without much hassle, and complex enough to do something interesting with.

In [ ]:
```
# let's download the MNIST data, if you do this locally and you downloaded
# you can change data paths to point to your existing files
train_dataset = dsets.MNIST(root='./MNISTdata',
                            train=True,
                            transform=transforms.ToTensor(),
                            download=True)

test_dataset = dsets.MNIST(root='./MNISTdata',
                           train=False,
                           transform=transforms.ToTensor())
```

Let's look at the digits and their labels

In [ ]:
```
ix=129
x,y = train_dataset[ix]
plt.imshow(x.squeeze().numpy())
print('label: y={}'.format(y))
```

Now let's define the dataloaders and train simple neural network like before. You'll recognize that the core is exactly the same: we do a forward pass, compute a loss, backpropagate the loss to compute the gradients, then let the optimizer update the weights.

In [ ]:
```
# The neural network hyperparameters.
input_size    = 784   # The MNIST image size = 28 x 28 = 784
hidden_size   = 100   # The number of nodes at the hidden layer
num_classes   = 10    # The number of output classes. In this case, from 0
num_epochs    = 5     # The number of times entire dataset is trained
batch_size    = 100   # The number of samples per minibatch
learning_rate = 1.0   # SGD step size
```

In [ ]:
```
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
```

In [ ]:
```
len(train_loader)
```

In [ ]:
```
#  define simple MLP network. Train network
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()                    # Inherited from the
        self.fc1 = nn.Linear(input_size, hidden_size)  # 1st Full-Connected
        self.relu = nn.ReLU()                          # Non-Linear ReLU La
        self.fc2 = nn.Linear(hidden_size, num_classes) # 2nd Full-Connected

    def forward(self, x):                              # Forward pass: stac
        x = x.view(x.size(0), -1) # flatten (bs x 1 x 28 x 28) -> (bs x 784
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
def test(net, dl):
    right, tot = 0, 0
    net.eval() # Set dropout and possibly other modules in eval mode.
    for x,y in dl:
        ypred = net(x).argmax(dim=1) # select index of maximal score
        right += (ypred == y).sum().item()
        tot   += x.size(0)
    return 1.* right / tot
```

In [ ]:
```
device = torch.device('cpu') # if on gpu-enabled machine, set torch.device(
# create the net based on this class definition
net = Net(input_size, hidden_size, num_classes).to(device)
# define the optimizer
optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0
print('Before training: {:.1f}% test accuracy'.format(100*test(net, test_lo
for epoch in range(num_epochs):
    scheduler.step()
    for i, (x,y_target) in enumerate(train_loader):
        y_probs = F.log_softmax(net(x), dim=1)
        output = F.nll_loss(y_probs, y_target)
        output.backward()
        optimizer.step()
        net.zero_grad()
    print('After epoch {}: {:.1f}% test accuracy'.format(epoch, 100*test(ne
print('End of training: {:.1f}% train accuracy'.format(100*test(net, train_
```

some questions for you to investigate:

- what does softmax do? (test on a random vector)
- what is its purpose? (read the docs)
- what does nll_loss do? can you manually compute it?

In [ ]:
```
# show the prediction on some samples.
ix=1234
x,y = test_dataset[ix]
plt.imshow(x.squeeze().numpy())
print('Ground truth label: y={}'.format(y))
y_probs = F.softmax(net(x), dim=1)
print ('Model probabilities: ')
print(' / '.join(['{}: {:.3f}'.format(k,v)
                  for k,v in zip(range(10), y_probs.squeeze().tolist()) ] )
print('Model prediction: ', y_probs.argmax(1))
```

Now we used a simple flat neural network which looks at the image as a flat vector, without awareness of the 2D structure or which pixels neighbor each other. A convolutional neural network is an architecture that takes the 2D structure of the image into account by sliding a kernel over all the different locations in the image. This kind of neural network has been very succesful in image recognition [1] and speech recognition [2,3]. Pytorch and other deep learning toolboxes are designed to deal with this kind of data and with convolutional neural networks just as easily as with flat data. Try swapping out the network above for a convolutional neural network, see for example the pytorch tutorial [4].

[1] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks (https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks) [2] http://www.cs.toronto.edu/~asamir/papers/icassp13_cnn.pdf (http://www.cs.toronto.edu/~asamir/papers/icassp13_cnn.pdf) [3] https://arxiv.org/abs/1509.08967 (https://arxiv.org/abs/1509.08967) [4] https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py) [5] https://colab.research.google.com/drive/1jxUPzMsAkBboHMQtGyfv5M5c7hU8Ss2c (https://colab.research.google.com/drive/1jxUPzMsAkBboHMQtGyfv5M5c7hU8Ss2c)

In [ ]:
```
# have fun
```

Finishing notes

Inspiration for this lab and the lecture:

- An old lab I made in lua torch https://github.com/tomsercu/torchtutorial (https://github.com/tomsercu/torchtutorial)
- This pytorch intro notebook https://colab.research.google.com/drive/1jxUPzMsAkBboHMQtGyfv5M5c7hU8Ss2c (https://colab.research.google.com/drive/1jxUPzMsAkBboHMQtGyfv5M5c7hU8Ss2c)
- The official pytorch tutorial https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)
- Yann LeCuns deep learning course in 2015 https://cilvr.nyu.edu/doku.php?id=deeplearning2015:schedule (https://cilvr.nyu.edu/doku.php?id=deeplearning2015:schedule)

In [ ]: