

THE DATA SCIENCE LAB

Neural Regression Using PyTorch: Training

03/03/2021

[GET CODE DOWNLOAD](#)

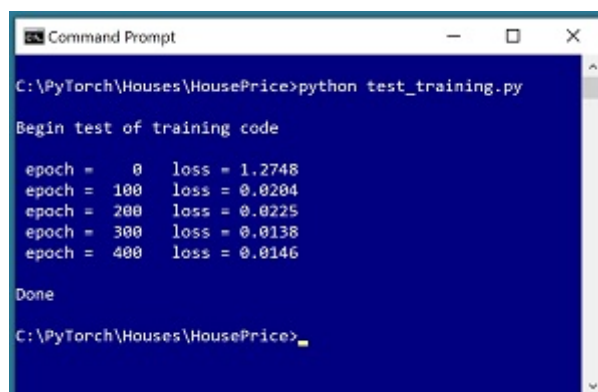
The training demo program begins execution with:

```
T.manual_seed(1)
np.random.seed(1)
train_file = ".\\Data\\houses_train.txt"
train_ds = HouseDataset(train_file, m_rows=200)
```

MOST POPULAR

The global PyTorch and NumPy random number generator seeds are set so that results will be reproducible. Unfortunately, due to multiple threads of execution, in some cases your results will not be reproducible even if you set the seed values.

The demo assumes that the training data is located in a subdirectory named Data. The HouseDataset object reads all 200 training data items into memory. If your training data size is very large you can read just part of the data into memory using the m_rows (maximum rows) parameter.



```
Command Prompt
C:\PyTorch\Houses\HousePrice>python test_training.py
Begin test of training code
epoch = 0    loss = 1.2748
epoch = 100  loss = 0.0204
epoch = 200  loss = 0.0225
epoch = 300  loss = 0.0138
epoch = 400  loss = 0.0146
Done
C:\PyTorch\Houses\HousePrice>
```

[Click on image for larger view.]

Figure 2: Testing the Training Code

The demo program prepares training with these statements:

```
bat_size = 10
train_ldr = T.utils.data.DataLoader(train_ds,
    batch_size=bat_size, shuffle=True)
net = Net().to(device)
net.train() # set mode
```

The training data loader is configured to read batches of 10 items at a time. In theory, the batch size doesn't matter too much, but in practice the batch size greatly affects how quickly training works. When you have a choice, it makes sense to use a batch size that divides the dataset size evenly so that all batches have the same size. Because the demo test House data has 200 rows, there will be 20 batches of 10 items each and no leftover items. It is very important to set `shuffle=True` when training because the default value is `False` which will often result in failed training.

After the neural network is created, it is set into training mode using the statement `net.train()`. If your neural network has a dropout layer or a batch normalization layer, you must set the network to `train()` mode during training and to `eval()` mode when using the network at any other time, such as making a prediction or computing model classification accuracy. The default state is `train()` mode so setting the mode isn't necessary for the demo network for two reasons: it's already in `train()` mode, and it doesn't use dropout or batch normalization. However, in my opinion it's good practice to always explicitly set the network mode.

The `train()` mode method works by reference so you can write just `net.train()` instead of `net = net.train()`. Note that the statement `net.train()` looks like it's an instruction to train a net object, but that is not what's happening.

The demo continues training preparation with these three statements:

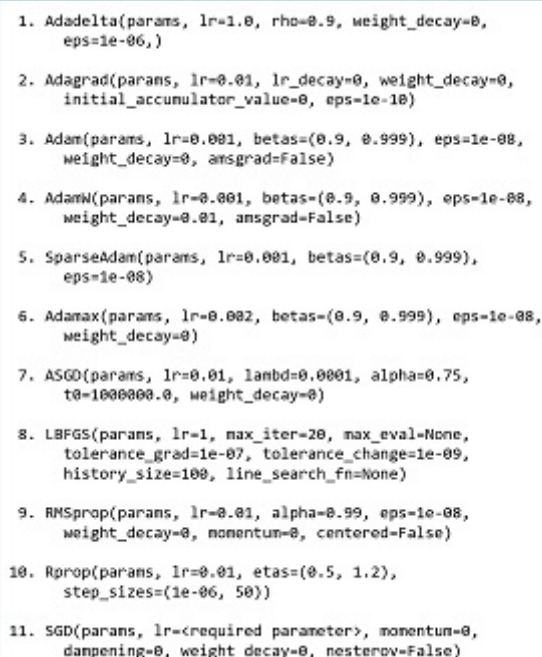
```
lrn_rate = 0.005
loss_func = T.nn.MSELoss()
optimizer = T.optim.Adam(net.parameters(),
    lr=lrn_rate)
```

For regression problems, the most common loss (error) function is `MSELoss` (mean squared error loss). In some rare cases you can use `BCELoss` (binary cross entropy) but only if all

target values in the training data have been normalized to a $[0, 1]$ range, and all computed output values are scaled to a $[0, 1]$ range using logistic sigmoid activation.

The demo program uses the Adam ("adaptive momentum") training optimizer. Adam often works better than basic SGD ("stochastic gradient descent") for regression problems.

PyTorch 1.7 supports 11 different training optimization techniques. Understanding all the details of PyTorch optimizers is difficult. Each technique's method has several parameters which are quite complex and which often have a dramatic effect on training performance. See the list in **Figure 3**.



```

1. Adadelta(params, lr=1.0, rho=0.9, weight_decay=0,
   eps=1e-06,)
2. Adagrad(params, lr=0.01, lr_decay=0, weight_decay=0,
   initial_accumulator_value=0, eps=1e-10)
3. Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
   weight_decay=0, amsgrad=False)
4. AdamW(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
   weight_decay=0.01, amsgrad=False)
5. SparseAdam(params, lr=0.001, betas=(0.9, 0.999),
   eps=1e-08)
6. Adamax(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08,
   weight_decay=0)
7. ASGD(params, lr=0.01, lambd=0.0001, alpha=0.75,
   t0=1000000.0, weight_decay=0)
8. LBFGS(params, lr=1, max_iter=20, max_eval=None,
   tolerance_grad=1e-07, tolerance_change=1e-09,
   history_size=100, line_search_fn=None)
9. RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08,
   weight_decay=0, momentum=0, centered=False)
10. Rprop(params, lr=0.01, etas=(0.5, 1.2),
   step_sizes=(1e-06, 50))
11. SGD(params, lr=<required parameter>, momentum=0,
   dampening=0, weight_decay=0, nesterov=False)

```

[Click on image for larger view.]

Figure 3: PyTorch Optimizers

Fortunately, almost all of the PyTorch optimizer parameters have reasonable default values. As a general rule of thumb, for regression problems I start by trying Adam with default parameter values. Then if I don't get good results, I'll adjust the learning rate parameter. If training still doesn't work, I'll try SGD.

A learning rate controls how much a network weight or bias changes on each update during training. For Adam it's best to use a small initial learning rate because the algorithm can dynamically change the learning rate during training.

The key takeaway is that if you're new to PyTorch you could easily spend weeks exploring the nuances of different training optimizers and never get any programs written. Optimizers are important but it's better to learn about different optimizers by experimenting with them slowly

over time, with different problems, than it is to try and master all their details before writing any code.

After training has been prepared, the demo program starts the training:

```
for epoch in range(0, 500):
    # T.manual_seed(1 + epoch) # recovery reproduce
    epoch_loss = 0.0 # sum avg loss per item

    for (batch_idx, batch) in enumerate(train_loader):
        X = batch[0] # predictors shape [10,8]
        Y = batch[1] # targets shape [10,1]
        optimizer.zero_grad()
        oupt = net(X) # shape [10,1]
    . . .
```

Setting the manual seed inside the main training loop is necessary if you periodically save the model's weights and biases as checkpoints, so that if the training machine crashes during training, you can recover with the exact same state. Because the demo program doesn't save checkpoints, it's not necessary to set the seed, which is why that statement is commented out.

For each batch of 10 items, the sets of 8 predictor values are extracted as X and the 10 target house price values are extracted as Y. The inputs are fed to the network and the results are captured as oupt. The zero_grad() method resets the gradients of all weights and biases so that new gradients can be computed and used to update the weights and biases.

The demo continues with:

```
loss_val = loss_func(oupt, Y) # avg loss in batch
epoch_loss += loss_val.item() # a sum of averages
loss_val.backward() # compute gradients
optimizer.step() # update weights
```

It's important to monitor the error/loss during training so that you can tell if training is working or not. There are three main ways to monitor loss. An epoch_loss value is computed for each batch of input values. This batch loss value is the average of the loss values for each item in the batch. For example, if a batch has four items and the MSE loss values for each of the four items are (2.00, 7.00, 3.00, 6.00) then the computed batch loss is $18.00 / 4 = 4.50$. The simplest approach is to just display the loss for either the first batch or the last batch for each

training epoch. It's usually not feasible to print the loss value for every batch because there are just too many batches processed in almost all realistic scenarios.

A second approach for monitoring loss during training is to accumulate each batch loss value, and then after all the batches in one epoch have been processed in one epoch, you can display the sum of the batch losses. For example, if one epoch consists of 3 batches of data and the batch average MSE loss values are (2.50, 5.10, 1.30) then the sum of the batch losses is $2.50 + 5.10 + 1.30 = 8.90$. This approach for monitoring loss is the one used by the demo program.

A third approach for monitoring loss is to compute an average loss per item for all training items. This is a bit tricky. First you would "un-average" the average loss value returned by the `loss_func()` method, by multiplying by the number of items in the batch, and then you'd accumulate the individual loss values. After all batches have been processed, you can compute an average loss per item over the entire dataset by dividing by the total number of training items. The code could look like this:

```
for epoch in range(0, 500):
    sum_epoch_loss = 0.0

    for (batch_idx, batch) in enumerate(train_loader):
        . . .
        loss_val = loss_func(output, Y) # avg loss in batch
        sum_vals = loss_val.item() * len(X) # "un-average"
        sum_epoch_loss += sum_vals          # accumulate
        . . .

    if epoch % 50 == 0:
        avg_loss = sum_epoch_loss / len(train_ds) # average
        print(avg_loss)
```

MOST POPULAR

None of the three approaches for monitoring loss during training give values that are easy to interpret. The important thing is to watch the values to see if the loss values are decreasing. It is possible for training loss to values to bounce around a bit, where a loss value might increase briefly, especially if your batch size is small. Because there are many ways to monitor and display cross entropy loss for neural regression, loss values usually can't be compared for different systems unless you know the systems are computing and displaying loss in the exact same way.

The `item()` method is used when you have a tensor that has a single numeric value. The `item()` method extracts the single value from the associated tensor and returns it as a regular scalar

value. Somewhat unfortunately (in my opinion), PyTorch 1.7 allows you to skip the call to `item()` so you can write the shorter `epoch_loss += loss_val` instead. Because `epoch_loss` is a non-tensor scalar, the interpreter will figure out that you must want to extract the value in the `loss_val` tensor. You can think of this mechanism as similar to implicit type conversion. However, the shortcut form without calling the `item()` method is slightly misleading in my opinion and so I use `item()` in most situations, even when it's not technically necessary.

The `loss_val` is a tensor that is the last value in the behind-the-scenes computational graph that represents the neural network being trained. The `loss_val.backward()` method uses the back-propagation algorithm to compute all the gradients associated with the weights and biases that a part of the network containing `loss_val`. Put another way, `loss_val.backward()` computes the gradients of the output node weights and bias, and then the `hid2` layer gradients, and then the `hid1` layer gradients.

The `optimizer.step()` statement uses the newly computed gradients to update all the weights and biases in the neural network so that computed output values will get closer to the target values. When you instantiate an optimizer object for a neural network, you must pass in the network parameters object and so the optimizer object effectively has full access to the network and can modify it.

The demo program concludes training with these statements:

MOST POPULAR

```

. . .
    optimizer.step()

    if epoch % 100 == 0:
        print("epoch = %4d    loss = %0.4f" % \
              (epoch, epoch_loss))
        # TODO: save checkpoint

print("Done ")

```

After all batches have been processed, a training epoch has been completed and program execution exits the innermost for-loop. Although it's possible to display the accumulated loss value for every epoch, in most cases that's too much information and so the demo just displays the accumulated average loss once every 100 epochs. In some problem scenarios you might want to store all accumulated epoch loss values in memory, and then save them all to a text file after training completes. This allows you to analyze training without slowing it down.

Because training a neural network can take hours, days, or even longer, in all non-demo scenarios you'd want to periodically save training checkpoints. This topic will be explained in

the next article in this series.

Wrapping Up

Training a PyTorch neural network for a regression problem is simple and complicated at the same time. Training in PyTorch works at a low level. This requires a lot of effort but gives you maximum flexibility. The behind-the-scenes details and options such as optimizer parameters are very complex. But the good news is that the demo training code presented in this article can be used as a template for most of the neural regression problems you're likely to encounter.

Monitoring mean squared error/loss during training allows you to determine if training is working, but loss isn't a good way to evaluate a trained model. Ultimately prediction accuracy is the metric that's usually most important. Computing model accuracy, and saving a trained model to file, are topics in the next article in this series.

« previous 1 | 2

MOST POPULAR

About the Author

Dr. James McCaffrey works for Microsoft Research in Redmond, Wash. He has worked on several Microsoft products including Azure and Bing. James can be reached at jamccaff@microsoft.com.

PRINTABLE FORMAT

ALSO ON VISUAL STUDIO MAGAZINE

Microsoft's WinUI Wed with Uno Platform ...

a month ago • 3 comments

Microsoft and Uno Platform have teamed up to highlight the cross-platform app ...

What's Cool in C# 8 and .NET Core 3 -- ...

19 days ago • 59 comments

You're missing out on some cool features if you're building applications in ...

.NET 5 Blazor Powers 'Rock, Paper, ...

a month ago • 2 comments

A Microsoft project demonstrates a .NET 5 Blazor upgrade by ...

VS C New

a mor

Micro
Studi
add

[Comments](#) [Community](#) [Privacy Policy](#) [Login](#) 1[Recommend](#) 2 [Tweet](#) [Share](#) [Sort by Best](#)

LOG IN WITH

OR SIGN UP WITH DISQUS [?](#)

Be the first to comment.

[Subscribe](#) [Add Disqus to your site](#) [Add Disqus](#) [Add](#)

Featured

MOST POPULAR

Survey Reveals Bigger C# Community, Most and Least Popular Uses

Polling more than 19,000 developers, the new "Developer Economics State of the Developer Nation, 20th Edition," report is out, finding that C# has ticked up a notch in popularity, overtaking PHP for No. 5 on that ranking. What's more, the big twice-yearly report identifies what areas are most and least popular for coding in Microsoft's flagship programming language.



'Epic Fail': ASP.NET PM Struggles with Blazor Hot Reload in Live Demo

"Epic fail," commented a developer who this week tuned in to a livestreamed ASP.NET Community Standup event on "ASP.NET Core updates in .NET 6 Preview 3" in which Daniel Roth, principal program manager (the head guy) for ASP.NET, struggled with a Blazor Hot Reload demo.



Unity Game Platform Details Plans for .NET and C#

Unity Technologies, known for its real-time development platform used widely for gaming apps, has detailed its plans for incorporating new changes in .NET and C# being pushed out by Microsoft.



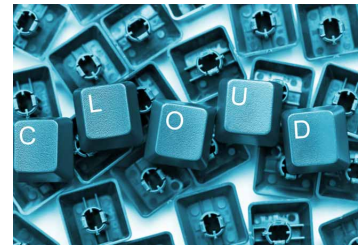
Red Hat Adds Java Features to Visual Studio Code

Red Hat's contribution to the Java Extension Pack adds type hierarchy and package refactoring when a file is moved.



Azure SDK Gets Communications Services Libraries, Based on Teams Tech

They empower cloud developers to create applications that incorporate chat, voice calling, video calling, traditional telephone calling, SMS messaging and other real-time communication functionality.



MOST POPULAR

.NET Insight

Sign up for our newsletter.

Email Address*

Country*

I agree to this site's **Privacy Policy**



Please type the letters/numbers you see above.

SUBMIT

Most Popular Articles

What's the Most Popular Component in the .NET/C# Tech Stack?

Survey Reveals Bigger C# Community, Most and Least Popular Uses

'Epic Fail': ASP.NET PM Struggles with Blazor Hot Reload in Live Demo

Hundreds of Developers Sound Off on Visual Studio 2022

Visual Studio 2022: Faster, Leaner and 64-bit (More Memory!)

MOST POPULAR

Free Webcasts

What's Ahead for .NET Development in 2021: Half-Day Virtual Summit

The MSIX Journey: What have we learned?

Hottest Third Party .NET Tools

How Blazor Changes Web Development

> More Webcasts

Upcoming Events

VSLive! 2-Day Seminar: Build A .NET 5 MVC App

May 4-5, 2021 [Virtual]

VSLive! 2-Day Seminar: Develop Secure ASP.NET Apps

May 20-21, 2021 [Virtual]

VSLive! 1-Day Workshop: Dependency Injection (DI) for the Developer

June 2, 2021 [Virtual]

VSLive! 2-Day Seminar: Learn Cloud-Native .NET Code

June 22-23, 2021 [Virtual]

VSLive! 4-Day Seminar: Hands-on Agile Software Dev

September 20-23, 2021

Austin, TX + Virtual

MOST POPULAR

VSLive! 2-Day Seminar: Learn Web APIs with ASP.NET

September 28-29, 2021 [Virtual]

VSLive! 2-Day Seminar: DevSecOps - Security In App Delivery

October 19-20, 2021 [Virtual]

Live! 360 Orlando 6-Day Conference

November 14-19, 2021

Orlando, FL

VSLive! 2-Day Seminar: ASP.NET Core 6 Service and Website

December 13-14, 2021 [Virtual]

Application Development Trends

[AWSInsider.net](#)

[Enterprise Systems](#)

[FutureTech360](#)

[Live! 360](#)

[MCPmag.com](#)

[Prophyts](#)

[Pure AI](#)

[Redmond](#)

[Redmond Channel Partner](#)

[TechMentor Events](#)

[Virtualization & Cloud Review](#)

[Visual Studio Live!](#)



© 1996-2021 **1105 Media Inc.** See our **Privacy Policy**, **Cookie Policy** and **Terms of Use**. **CA: Do Not Sell My Personal Info**

Problems? Questions? Feedback? E-mail us.

MOST POPULAR