THE DATA SCIENCE LAB

Neural Regression Using PyTorch: Defining a Network

Dr. James McCaffrey of Microsoft Research presents the second of four machine learning articles that detail a complete end-to-end production-quality example of neural regression using PyTorch.

By James McCaffrey 02/11/2021

GET CODE DOWNLOAD

MOST POPULAR

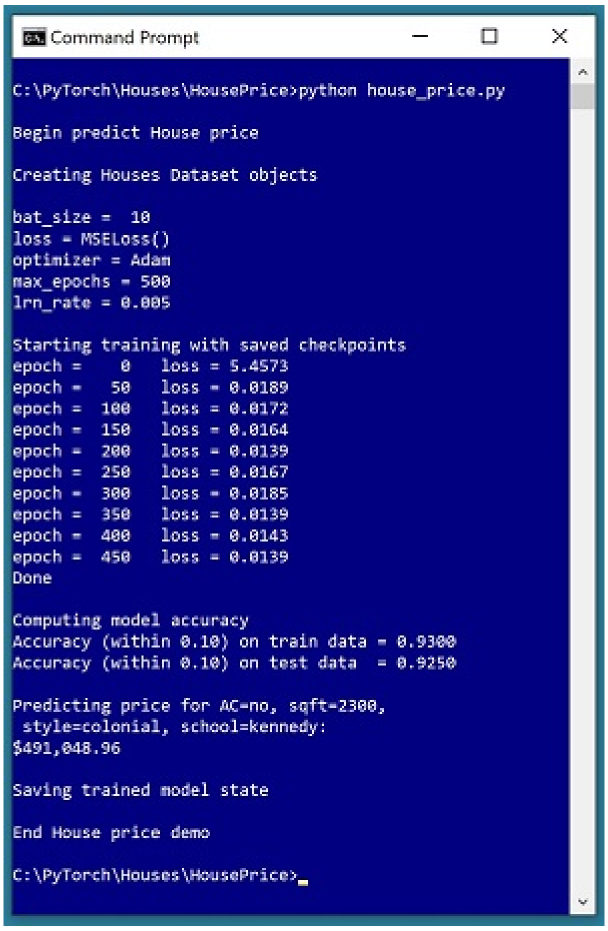
The goal of a regression problem is to predict a single numeric value, for example, predicting the price of a used car based on variables such as mileage, brand and year manufactured. There are several classical statistics techniques for regression problems. Neural regression solves a regression problem using a neural network. This article is the second in a series of four articles that present a complete end-to-end production-quality example of neural regression using PyTorch. The recurring example problem is to predict the price of a house based on its area in square feet, air conditioning (yes or no), style ("art_deco," "bungalow," "colonial") and local school ("johnson," "kennedy," "lincoln").

The process of creating a PyTorch neural network for regression consists of six steps:

1. Prepare the training and test data
2. Implement a Dataset object to serve up the data in batches
3. Design and implement a neural network
4. Write code to train the network
5. Write code to evaluate the model (the trained network)
6. Write code to save and use the model to make predictions for new, previously unseen data

Each of the six steps is complicated. And the six steps are tightly coupled which adds to the difficulty. This article covers the third step -- designing and implementing a neural network for neural regression.

A good way to see where this series of articles is headed is to take a look at the screenshot of the demo program in **Figure 1** . The demo begins by creating Dataset and DataLoader objects which have been designed to work with the house data. Next, the demo creates an 8-(10-10)-1 deep neural network. The demo prepares training by setting up a loss function (mean squared error), a training optimizer function (Adam) and parameters for training (learning rate and max epochs).



```
C:\PyTorch\Houses\HousePrice>python house_price.py

Begin predict House price

Creating Houses Dataset objects

bat_size =  10
loss = MSELoss()
optimizer = Adam
max_epochs = 500
lrn_rate = 0.005

Starting training with saved checkpoints
epoch =    0   loss = 5.4573
epoch =   50   loss = 0.0189
epoch =  100   loss = 0.0172
epoch =  150   loss = 0.0164
epoch =  200   loss = 0.0139
epoch =  250   loss = 0.0167
epoch =  300   loss = 0.0185
epoch =  350   loss = 0.0139
epoch =  400   loss = 0.0143
epoch =  450   loss = 0.0139
Done

Computing model accuracy
Accuracy (within 0.10) on train data = 0.9300
Accuracy (within 0.10) on test data  = 0.9250

Predicting price for AC=no, sqft=2300,
 style=colonial, school=kennedy:
$491,048.96

Saving trained model state

End House price demo

C:\PyTorch\Houses\HousePrice>_
```

[Click on image for larger view.]

Figure 1: Neural Regression in Action

The demo trains the neural network for 500 epochs in batches of 10 items. An epoch is one complete pass through the training data. The training data has 200 items, therefore, one training epoch consists of processing 20 batches of 10 training items.

During training, the demo computes and displays a measure of the current error (also called loss) every 50 epochs. Because error slowly decreases, it appears that training is succeeding. Behind the scenes, the demo program saves checkpoint information after every 50 epochs so that if the training machine crashes, training can be resumed without having to start from the beginning.

这个使用checkpoints的逻辑思维已经不止在一个地方出现的了，这个是一个非常好的逻辑思维，可以借用（虽然我现在也不清楚，怎么样子resume training data）

After training the network, the demo program computes the prediction accuracy of the model based on whether or not the predicted house price is within 10 percent of the true house price. The accuracy on the training data is 93.00 percent (186 out of 200 correct) and the accuracy on the test data is 92.50 percent (37 out of 40 correct). Because the two accuracy values are similar, it is likely that model overfitting has not occurred.

Next, the demo uses the trained model to make a prediction on a new, previously unseen house. The raw input is (air conditioning = "no", square feet area = 2300, style = "colonial", school = "kennedy"). The raw input is normalized and encoded as (air conditioning = -1, area = 0.2300, style = 0,0,1, school = 0,1,0). The computed output price is 0.49104896 which is equivalent to $491,048.96 because the raw house prices were all normalized by dividing by 1,000,000.

他这里就是对data进行normalisation的方式是需要学习的了，因为我自己现在对于这个还是没有太多的经验了。另外一个方式，就是请教其他人了。

The demo program concludes by saving the trained model using the state dictionary approach. This is the most common of three standard techniques.

这个也是一个非常好的点，也是需要搞清楚的了。

This article assumes you have an intermediate or better familiarity with a C-family programming language, preferably Python, but doesn't assume you know very much about PyTorch. The complete source code for the demo program, and the two data files used, are available in the download that accompanies this article. All normal error checking code has been omitted to keep the main ideas as clear as possible.

MOST POPULAR

To run the demo program, you must have Python and PyTorch installed on your machine. The demo programs were developed on Windows 10 using the Anaconda 2020.02 64-bit distribution (which contains Python 3.7.6) and PyTorch version 1.7.0 for CPU installed via pip. You can find detailed step-by-step installation instructions for this configuration in my **blog post** .

这里需要注意一下，这个原始代码，就是因为path的原因，所以没有办法正常运行的了，一旦修改了path之后，这个代码就是可以正常的运行的了。

The House Data

The raw House data is synthetic and was generated programmatically. There are a total of 240 data items, divided into a 200-item training dataset and a 40-item test dataset. The raw data looks like:

```
no     1275   bungalow   $318,000.00   lincoln
yes    1100   art_deco   $335,000.00   johnson
no     1375   colonial   $286,000.00   kennedy
yes    1975   bungalow   $512,000.00   lincoln
. . .
no     2725   art_deco   $626,000.00   kennedy
```

Each line of tab-delimited data represents one house. The value to predict, house price, is in 0-based column [3]. The predictors variables in columns [0], [1], [2] and [4] are air conditioning, area in square feet, style and local school. For simplicity, there are just three house styles and three schools.

House area values were normalized by dividing by 10,000 and house prices were normalized by dividing by 1,000,000. Air conditioning was binary encoded as no = -1, yes = +1. Style was one-hot encoded as "art_deco" = (1,0,0), "bungalow" = (0,1,0), "colonial" = (0,0,1). School was one-hot encoded as "johnson" = (1,0,0), "kennedy" = (0,1,0), "lincoln" = (0,0,1). The resulting normalized and encoded data looks like:

```
-1   0.1275   0 1 0   0.3180   0 0 1
 1   0.1100   1 0 0   0.3350   1 0 0
-1   0.1375   0 0 1   0.2860   0 1 0
 1   0.1975   0 1 0   0.5120   0 0 1
. . .
-1   0.2725   1 0 0   0.6260   0 1 0
```

ToDo: 一旦这个代码就是掌握了之后，我就是需要向boss要数据的了，就是需要看看我到底是需要哪一种类型的数据的了，因为他给的数据是不够的，就是非常的少，没有办法做这件事情的了。

After the structure of the training and test files was established, I designed and coded a PyTorch Dataset class to read the house data into memory and serve the data up in batches using a PyTorch DataLoader object. A Dataset class definition for the normalized encoded House data is shown in **Listing 1**.

MOST POPULAR

Listing 1: A Dataset Class for the Student Data

```
class HouseDataset(T.utils.data.Dataset):
  # AC  sq ft   style  price   school
  # -1  0.2500  0 1 0  0.5650  0 1 0
  #  1  0.1275  1 0 0  0.3710  0 0 1
  # air condition: -1 = no, +1 = yes
  # style: art_deco, bungalow, colonial
  # school: johnson, kennedy, lincoln

  def __init__(self, src_file, m_rows=None):
    all_xy = np.loadtxt(src_file, max_rows=m_rows,
      usecols=[0,1,2,3,4,5,6,7,8], delimiter="\t",
      comments="#", skiprows=0, dtype=np.float32)

    tmp_x = all_xy[:,[0,1,2,3,4,6,7,8]]
    tmp_y = all_xy[:,5].reshape(-1,1)    # 2-D
```

Preparing data and defining a PyTorch Dataset is not trivial. You can find the article that explains how to create Dataset objects and use them with DataLoader objects **here** .

The Overall Program Structure

The overall structure of the PyTorch neural regression program, with a few minor edits to save space, is shown in **Listing 2** . I prefer to indent my Python programs using two spaces rather than the more common four spaces.

Listing 2: The Structure of the Demo Program

```
# house_price.py
# PyTorch 1.7.0-CPU Anaconda3-2020.02
# Python 3.7.6 Windows 10

import numpy as np
import time
import torch as T
device = T.device("cpu")

class HouseDataset(T.utils.data.Dataset):
  # AC   sq ft   style  price   school
  # -1  0.2500  0 1 0  0.5650  0 1 0
  #  1  0.1275  1 0 0  0.3710  0 0 1
  # air condition: -1 = no, +1 = yes
  # style: art_deco, bungalow, colonial
  # school: johnson, kennedy, lincoln
```

MOST POPULAR

It's important to document the versions of Python and PyTorch being used because both systems are under continuous development. Dealing with versioning incompatibilities is a significant headache when working with PyTorch and is something you should not underestimate. The demo program imports the Python time module to timestamp saved checkpoints.

I prefer to use "T" as the top-level alias for the torch package. Most of my colleagues don't use a top-level alias and spell out "torch" many times per program. Also, I use the full form of submodules rather than supplying aliases such as "import torch.nn.functional as functional." In my opinion, using the full form is easier to understand and less error-prone than using many aliases.

The demo program defines a program-scope CPU device object. I usually develop my PyTorch programs on a desktop CPU machine. After I get that version working, converting to a CUDA

GPU system only requires changing the global device object to T.device("cuda") plus a minor amount of debugging.

The demo program sets the NumPy and PyTorch random number generator seed values so that program runs will be reproducible. The seed value of 4 was used only because it gives representative results.

The demo program defines just one helper method, accuracy(). All of the rest of the program control logic is contained in a main() function. It is possible to define other helper functions such as train_net(), evaluate_model(), and save_model(), but in my opinion this modularization approach makes the program more difficult to understand rather than easier to understand.

Defining a Neural Network for Neural Regression

The first step when designing a PyTorch neural network class for a regression problem is to determine its architecture. Neural architecture design includes the number of input and output nodes, the number of hidden layers and the number of nodes in each hidden layer, the activation functions for the hidden and output layers, and the initialization algorithms for the hidden and output layer nodes.

The number of input nodes is determined by the number of predictor values (after normalization and encoding), eight in the case of the House data. For most regression problems, there is just one output node, which holds the numeric value to predict. It is possible for a neural regression system to have two or more numeric values, but these problems are quite rare.

MOST POPULAR

The demo network uses two hidden layers, each with 10 nodes, resulting in an 8-(10-10)-1 network. The number of hidden layers and the number of nodes in each layer are hyperparameters. Their values must be determined by trial and error guided by experience. The term "AutoML" is sometimes used for any system that programmatically, to some extent, tries to determine good hyperparameter values.

ToDo: 需要把这一篇文章过一下，让自己心中就是有个数的

More hidden layers and more hidden nodes are not always better. The Universal Approximation Theorem (sometimes called the Cybenko Theorem) says, loosely, that for any neural architecture with multiple hidden layers, there is an equivalent architecture that has just one hidden layer. For example, a neural network that has two hidden layers with 5 nodes each, is roughly equivalent to a network that has one hidden layer with 25 nodes.

其实，这段话，就是从某种程度上，验证了，I-ELM的可行性

The definition of class Net is shown in **Listing 3**. In general, most of my colleagues and I use the term "network" or "net" to describe a neural network before it's been trained, and the term "model" to describe a neural network after it's been trained. However, the two terms are often used interchangeably.

Listing 3: Neural Regression Neural Network Definition

```
class Net(T.nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.hid1 = T.nn.Linear(8, 10)  # 8-(10-10)-1
    self.hid2 = T.nn.Linear(10, 10)

    self.oupt = T.nn.Linear(10, 1)

    T.nn.init.xavier_uniform_(self.hid1.weight)
    T.nn.init.zeros_(self.hid1.bias)
    T.nn.init.xavier_uniform_(self.hid2.weight)
    T.nn.init.zeros_(self.hid2.bias)
    T.nn.init.xavier_uniform_(self.oupt.weight)
    T.nn.init.zeros_(self.oupt.bias)

  def forward(self, x):
```

The Net class inherits from torch.nn.Module which provides much of the complex behind-the-scenes functionality. The most common structure for a neural regression network is to define the network layers and their associated weights and biases in the __init__() method, and the input-output computations in the forward() method.

MOST POPULAR

The __init__() Method

The __init__() method begins by defining the demo network's three layers of nodes:

```
  def __init__(self):
    super(Net, self).__init__()
    self.hid1 = T.nn.Linear(8, 10)  # 8-(10-10)-1
    self.hid2 = T.nn.Linear(10, 10)
    self.oupt = T.nn.Linear(10, 1)
```

1 2 next »

PRINTABLE FORMAT

THE DATA SCIENCE LAB

Neural Regression Using PyTorch: Defining a Network

02/11/2021

GET CODE DOWNLOAD

The first statement invokes the __init__() constructor method of the Module class from which the Net class is derived. The next three statements define the two hidden layers and the single output layer. Notice that you don't explicitly define an input layer because no processing takes place on the input values. ToDo: 需要搞清楚， 为什么时使用这个linear的函数结构的了

The Linear() class defines a fully connected network layer. Because of this, some neural networks will name the layers as "fc1," "fc2," and so on. You can loosely think of each of the three layers as three standalone functions (even though they're actually class objects). Therefore, the order in which you define the layers doesn't matter. In other words, defining the three layers in this order:

MOST POPULAR

```
    self.hid2 = T.nn.Linear(10, 10) # hidden 2
    self.oupt = T.nn.Linear(10, 1)  # output
    self.hid1 = T.nn.Linear(8, 10)  # hidden 1
```

has no effect on how the network computes its output. However, it makes sense to define the networks layers in the order in which they're used when computing an output value.

The demo program initializes the network's weights and biases like so:

```
    T.nn.init.xavier_uniform_(self.hid1.weight)
    T.nn.init.zeros_(self.hid1.bias)
    T.nn.init.xavier_uniform_(self.hid2.weight)
    T.nn.init.zeros_(self.hid2.bias)
    T.nn.init.xavier_uniform_(self.oupt.weight)
    T.nn.init.zeros_(self.oupt.bias)
```

If a neural network with one hidden layer has "ni" input nodes, "nh" hidden nodes, and "no" output nodes, there are (ni * nh) weights connecting the input nodes to the hidden nodes, and there are (nh * no) weights connecting the hidden nodes to the output nodes. Each hidden node and each output node has a special weight called a bias, so there'd be (nh + no) biases. For example, a 4-5-3 neural network has (4 * 5) + (5 * 3) = 35 weights and (5 + 3) = 8 biases. The pattern extends to multiple hidden layers and so the 8-(10-10)-1 demo network has (8 * 10) + (10 * 10) + (10 * 1) = 190 weights and (10 + 10 + 1) = 21 biases.

这段话，如果是理解microsoft的教程的话，就是非常容易理解的了

Each layer has a set of weights which connect it to the previous layer. Therefore, self.hid1.weight is a matrix of weights from the input nodes to the nodes in the hid1 layer, self.hid2.weight is a matrix of weights from the hid1 nodes to the hid2 nodes, and self.oupt.weight is a matrix of weights from the hid2 nodes to the output nodes.

If you don't explicitly initialize the values of weights and biases, PyTorch will automatically initialize them using a default mechanism. But in my opinion it's good practice to explicitly initialize the values of a network's weights and biases, so that your results are reproducible. The demo uses xavier_uniform_() initialization on all weights, and it initializes all biases to 0. The xavier() initialization technique is called glorot() in some neural libraries, notably TensorFlow and Keras. Notice the trailing underscore character in the names of the initializers. This indicates the initialization method modifies its weight matrix argument in place by reference, rather than as a return value.

PyTorch 1.7 supports a total of 13 different initialization functions, such as uniform_(), normal_(), constant_() and dirac_(). For most neural regression problems, the uniform_() and xavier_uniform_() functions work well.

MOST POPULAR

Based on my experience, for relatively small neural networks, in some cases plain uniform_() initialization works better than xavier_uniform_() initialization. The uniform_() function requires you to specify a range. For example, the statement:

这里也是给出了这个uniform和xavier_uniform的区别的了。有一个大概的印象就是可以的了。

```
T.nn.init.uniform_(self.hid1.weight, -0.05, +0.05)
```

would initialize the hid1 layer weights to random values between -0.05 and +0.05. Although the xavier_uniform_() function was designed for deep neural networks with many layers and many nodes, it often works well with simple neural networks too, and it has the advantage of not requiring the two range parameters. This is because xavier_uniform_() computes the range values based on the number of nodes in the layer to which it is applied.

With a neural network defined as a class with no parameters as shown, you can instantiate a network object with a single statement:

```
 net = Net().to(device)
```

Somewhat confusingly for PyTorch beginners, there is an entirely different approach you can use to define and instantiate a neural network. This alternative approach uses the Sequential class to both define and create a network at the same time. The following code creates a neural network that's almost the same as the demo network:

```
net = T.nn.Sequential(
  T.nn.Linear(8,10),
  T.nn.ReLU(),
  T.nn.Linear(10,10),
  T.nn.ReLU(),
  T.nn.Linear(10,1),
).to(device)
```

Notice this approach doesn't use explicit weight and bias initialization so you'd be using whatever the current PyTorch version default initialization scheme is (default initialization has changed at least three times since the PyTorch 0.2 version). It is possible to explicitly apply weight and bias initialization to a Sequential network but the technique is a bit awkward.

When using the Sequential approach, you don't have to define a forward() method because one is automatically created for you. In almost all situations I prefer using the class definition approach over the Sequential technique. The class definition approach is lower level than the Sequential technique which gives you a bit more flexibility. Additionally, understanding the class definition approach is essential if you want to create complex neural architectures such as LSTMs, CNNs and Transformers.

The forward() Method

When using the class definition technique to define a neural network, you must define a forward() method that accepts input tensor(s) and computes output tensor(s). The demo program's forward() method is defined as:

```
def forward(self, x):
  z = T.relu(self.hid1(x))
  z = T.relu(self.hid2(z))
  z = self.oupt(z)  # no activation
  return z
```

The x parameter is a batch of one or more tensors. The x input is fed to the hid1 layer and then relu() activation function is applied and the result is returned as a new tensor z. The relu() function ("rectified linear unit") is one of 28 non-linear activation functions supported by PyTorch 1.7. For neural regression problems, two activation functions that usually work well are relu() and tanh(). Most of the 26 other activation functions, such as elu(), celu(), silu() and gelu(), are used primarily with specialized neural architectures.

In most neural regression problems, you don't apply an activation function to the output node. However, in some cases an output activation can be used. For example, if all dependent values to predict have been normalized so that they're between -1.0 and +1.0 then it'd be feasible to apply tanh() activation on the output node because tanh() returns values between -1.0 and +1.0. Or if all dependent values have been normalized to a range of [0.0, 1.0], such as the demo data, it'd be feasible to apply sigmoid() activation. To the best of my knowledge, there are no solid research results on the topic of output node activation for neural regression systems.

ToDo: 关于activation function在output node上的应用，还是需要留心一下，现在暂时就是不用了，

A mildly annoying characteristic of PyTorch is that there are often multiple variations of the same function. For example, there are at least three tanh() functions: torch.tanh(), torch.nn.Tanh() and torch.nn.functional.tanh(). Multiple versions of functions exist mostly because PyTorch is an open source project and its code organization has evolved somewhat organically over time. There is no easy way to deal with the confusion of multiple versions of some PyTorch functions.

ToDo: 这个问题也确实是让我头疼的，我也不太清楚有没有什么办法去解决这个问题的了，有空google一下的了

MOST POPULAR

Testing the Network

It's good practice to test a neural network before trying to train it. The short program in Listing 4 shows an example. The test program instantiates an 8-(10-10)-1 neural network as described in this article and then feeds it an input corresponding to (air conditioning = "no", square feet = 2500, style = "bungalow", school = "lincoln"). The computed output value is 0.07998378 (representing a price of $79,983.78). The predicted price isn't reasonable because the network hasn't been trained. See the screenshot in **Figure 2**.

Listing 4: Testing the Network

```
# test_net.py

import torch as T
device = T.device("cpu")

class Net(T.nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.hid1 = T.nn.Linear(8, 10)  # 8-(10-10)-1
    self.hid2 = T.nn.Linear(10, 10)
    self.oupt = T.nn.Linear(10, 1)

    T.nn.init.xavier_uniform_(self.hid1.weight)
    T.nn.init.zeros_(self.hid1.bias)
    T.nn.init.xavier_uniform_(self.hid2.weight)
    T.nn.init.zeros_(self.hid2.bias)
```

The key statements in the test program are:

```
net = Net().to(device)
net.eval()  # set mode
# AC=no, sqft=2500, style=bungalow, school=lincoln
x = T.tensor([[-1, 0.2500, 0,1,0, 0,0,1]],
      dtype=T.float32).to(device)
with T.no_grad():
  y = net(x)

print("input = ")
print(x)
print("output = ")
print("%0.8f" % y.item())
```

MOST POPULAR

The net object is instantiated as you might expect. Then the network is placed into eval() mode. The alternative is train() mode. The call to compute output is placed in a no_grad() block because no training is performed. Placing the network into evaluation mode and using a no_grad() block are both optional but good practice in my opinion. The output y is a PyTorch tensor with a single value. The single scalar value is extracted as a regular Python float value using the item() function.

If you're an experienced programmer but new to PyTorch, the call to the neural network seems to make no sense. Where is the forward() method? Why does it look like the Net object is being re-instantiated using the x tensor?

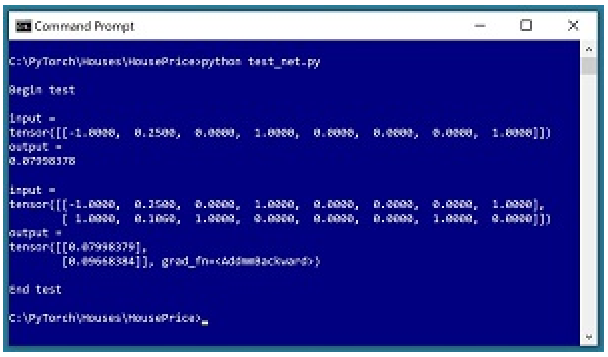
As it turns out, the net object inherits a special Python __call__() method from the torch.nn.Module class. Any object that has a __call__() method can invoke the method implicitly using simplified syntax of object(input). Additionally, if a PyTorch object which is derived from Module has a method named forward(), then the __call__() method calls the forward() method. To summarize, the statement y = net(x) invisibly calls the inherited __call__() method which in turn calls the program-defined forward() method. The implicit call mechanism may seem like a major hack but in fact there are good reasons for it.

ToDo: 这里就是没有看明白，等会儿过代码的时候，再过一遍

You can verify the calling mechanism by running this code:

```
y = net(x)
y = net.forward(x)  # same output
y = net.__call__(x) # same output
```

In non-exploration scenarios, you should not call a neural network using the __call__() or __forward__() methods because the implied call mechanism does necessary behind-the-scenes logging and other actions.



```
C:\PyTorch\Houses\HousePrice>python test_net.py

Begin test

input =
tensor([[-1.0000,  0.2500,  0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  1.0000]])
output =
0.07998378

input =
tensor([[-1.0000,  0.2500,  0.0000,  1.0000,  0.0000,  0.0000,  0.0000,  1.0000],
        [ 1.0000,  0.1060,  1.0000,  0.0000,  0.0000,  0.0000,  1.0000,  0.0000]])
output =
tensor([[0.07998379],
        [0.09668384]], grad_fn=<AddmmBackward>)

End test

C:\PyTorch\Houses\HousePrice>_
```

[Click on image for larger view.]
Figure 2: *Testing the Neural Network*

Notice the input x is a 2-dimensional matrix (indicated by the double square brackets) rather than a 1-dimensional vector because the network is expecting a batch of items as input. You could verify this by setting up a second input and calling it like so:

```
x = T.tensor([[-1, 0.2500, 0,1,0, 0,0,1],
              [+1, 0.1060, 1,0,0, 0,1,0]],
      dtype=T.float32).to(device)
y = net(x)
```

If you look at the screenshot in **Figure 2**, you'll notice that the output is displayed as:

```
tensor([[0.07998379],
        [0.09668384]], grad_fn=<AddmmBackward>)
```

Because the output was computed without using a no_grad() block, an associated gradient is computed with the output tensor. The grad_fn is the "gradient function" associated with the tensor. A gradient is needed by PyTorch for use in training. In fact, the ability of PyTorch to automatically compute gradients is arguably one of the library's two most important features (along with the ability to compute on GPU hardware).

To summarize, when calling a PyTorch neural network to compute output during training, you should set the mode as net.train() and not use the no_grad() statement. But when you aren't training, you should set the mode as net.eval() and use the no_grad() statement.

Wrapping Up

Defining a PyTorch neural network for regression is not trivial but the demo code presented in this article can serve as a template for most scenarios. In situations where a neural network model tends to overfit, you can use a technique called dropout. Model overfitting is characterized by a situation where model accuracy on the training data is good, but model accuracy on the test data is poor.

You can add a dropout layer after any hidden layer. For example, to add two dropout layers to the demo network, you could modify the __init__() method like so:

```
  def __init__(self):
    super(Net, self).__init__()
    self.hid1 = T.nn.Linear(8, 10)  # 8-(10-10)-1
    self.drop1 = T.nn.Dropout(0.50)
    self.hid2 = T.nn.Linear(10, 10)
    self.drop2 = T.nn.Dropout(0.25)
    self.oupt = T.nn.Linear(10, 1)
```

The first dropout layer will ignore 0.50 (half) of randomly selected nodes in the hid1 layer on each call to forward() during training. The second dropout layer will ignore 0.25 of randomly selected nodes in the hid2 layer during training. As I explained previously, these statements are class instantiations, not sequential calls of some sort, and so the statements could be defined in any order.

The forward() method would use the dropout layers like so:

```
  def forward(self, x):
    z = T.relu(self.hid1(x))
    z = self.drop1(z)
    z = T.relu(self.hid2(z))
```

```
        z = self.drop2(z)
        z = self.oupt(z)
        return z
```

Using dropout introduces randomness into the training which tends to make the trained model more resilient to new, previously unseen inputs. Because dropout is intended to control model overfitting, in most situations you define a neural network without dropout, and then add dropout only if overfitting seems to be happening.

« previous 1 | 2

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.

MOST POPULAR

PRINTABLE FORMAT

ALSO ON **VISUAL STUDIO MAGAZINE**

What Do Devs Want for VS 2022? The Top …
6 days ago • 6 comments
After Microsoft addressed a top developer feature request with this week's …

Microsoft's Surface Duo Dev Team …
2 months ago • 2 comments
Microsoft is helping Google provide support for foldable devices -- specifically the …

'Epic Fail': ASP.NET PM Struggles with …
4 days ago • 21 comments
"Epic fail," commented a developer who this week tuned in to a …

Micr Stuc
2 mor
Micrc enter coder