Neural Network Input¶

So now that you know the basics of what Pytorch is, let's apply it using a basic neural network example. The very first thing we have to consider is our data. In most tutorials, this bit is often overlooked in the interest of going straight to the training of a neural network. That said, as a programmer working with neural networks, one of your largest tasks is preprocessing your data and formatting it in such as way to be easiest for the neural network to work with.

First, we need a dataset.

We're just going to use data from Pytorch's "torchvision." Pytorch has a relatively handy inclusion of a bunch of different datasets, including many for vision tasks, which is what `torchvision` is for.

We're going to first start off by using Torchvision because you should know it exists, plus it alleviates us the headache of dealing with datasets from scratch.

That said, this is the probably the last time that we're going to do it this way. While it's nice to load and play with premade datasets, it's very rare that we get to do that in the real world, so it is essential that we learn how to start from a more raw dataset.

For now though, we're just trying to learn about how to do a basic neural network in pytorch, so we'll use `torchvision` here, to load the MNIST dataset, which is a image-based dataset showing handwritten digits from 0-9, and your job is to write a neural network to classify them.

To begin, let's make our imports and load in the data:

```
import torch
import torchvision
from torchvision import transforms, datasets

train = datasets.MNIST('', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor()
                       ]))

test = datasets.MNIST('', train=False, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor()
                       ]))
```

Above, we're just loading in the dataset, shuffling it, and applying any transforms/pre-processing to it.

Next, we need to handle for how we're going to iterate over that dataset:

```
trainset = torch.utils.data.DataLoader(train, batch_size=10, shuffle=True)
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=False)
```

You'll see later why this torchvision stuff is basically cheating! For now though, what have we done? Well, quite a bit.

Training and Testing data split
To train any machine learning model, we want to first off have training and validation datasets. This is so we can use data that the machine has never seen before to "test" the machine.

Shuffling
Then, within our training dataset, we generally want to randomly shuffle the input data as much as possible to hopefully not have any patterns in the data that might throw the machine off.

For example, if you fed the machine a bunch of images of zeros, the machine would learn to classify everything as zero. Then you'd start feeding it ones, and the machine would figure out pretty quick to classify everything as ones...and so on. Whenever you stop, the machine would probably just classify everything as the last thing you trained on. If you shuffle the data, your machine is much more likely to figure out what's what.

Scaling and normalization
Another consideration at some point in the pipeline is usually scaling/normalization of the dataset. In general, we want all input data to be between zero and one. Often many datasets will contain data in ranges that are not within this range, and we generally will want to come up with a way to scale the data to be within this range.

For example, an image is comprised of pixel values, most often in the range of 0 to 255. To scale image data, you usually just divide by 255. That's it. Even though all features are just pixels, and all you do is divide by 255 before passing to the neural network, this makes a *huge* difference.

Batches
Once you've done all this, you then want to pass your training dataset to your neural network.

Not so fast!

There are two major reasons why you can't just go and pass your entire dataset at once to your neural network:

1. Neural networks shine and outperform other machine learning techniques because of how well they work on big datasets. Gigabytes. Terabytes. Petabytes! When we're just learning, we tend to play with datasets smaller than a gigabyte, and we can often just toss the entire thing into the VRAM of our GPU or even more likely into RAM.

Unfortunately, in practice, you would likely not get away with this.

1. The aim with neural networks is to have the network *generalize* with the data. We want the neural network to actually learn general principles. That said, neural networks often have millions, or tens of millions, of parameters that they can tweak to do this. This means neural networks can also just memorize things. While we hope neural networks will generalize, they often learn to just memorize the input data. Our job as the scientist is to make it as hard as possible for the neural network to just memorize.

This is another reason why we often track "in sample" validation acccuracy and "out of sample" validation accuracy. If these two numbers are similar, this is good. As they start to diverge (in sample usually goes up considerably while out of sample stays the same or drops), this usually means your neural network is starting to just memorize things.

One way we can help the neural network to not memorize is, at any given time, we feed only some specific batch size of data. This is often something between 8 and 64.

Although there is no actual reason for it, it's a common trend in neural networks to use base 8 numbers for things like neuron counts, batch sizes...etc.

This batching helps because, with each batch, the neural network does a back propagation for new, updated weights with hopes of decreasing that loss.

With one giant passing of your data, this would include neuron changes that had nothing to do with general principles and were just brute forcing the operation.

By passing many batches, each with their own gradient calcs, loss, and backprop, this means each time the neural network optimizes things, it will sort of "keep" the changes that were actually useful, and erode the ones that were just basically memorizing the input data.

Given a large enough neural network, however, even with batches, your network can still just simply memorize.

This is also why we often try to make the smallest neural network as possible, so long as it still appears to be learning. In general, this will be a more successful model long term.

Now what?

Well, we have our data, but what is it really? How do we work with it? We can iterate over our data like so:

```python
for data in trainset:
    print(data)
    break
```

/

```
[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., 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., 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.],
          [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., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]]]), tensor([2, 3, 1, 2, 0, 8, 9, 6, 1,
```

Each iteration will contain a batch of 10 elements (that was the batch size we chose), and 10 classes. Let's just look at one:

```
X, y = data[0][0], data[1][0]
```

X is our input data. The features. The thing we want to predict. y is our label. The classification. The thing we hope the neural network can learn to predict. We can see this by doing.

`data[0]` is a bunch of features for things and `data[1]` is all the targets. So:

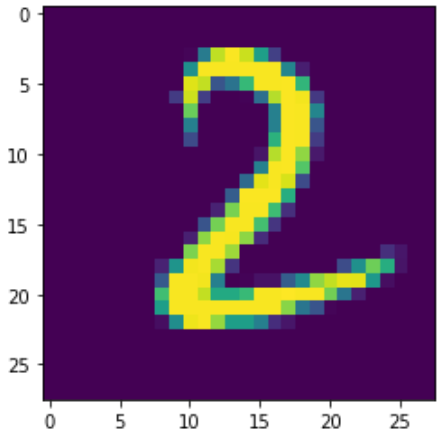
```
print(data[1])
```

```
tensor([2, 3, 1, 2, 0, 8, 9, 6, 1, 2])
```

As you can see, `data[1]` is just a bunch of labels. So, since `data[1][0]` is a 2, we can expect `data[0][0]` to be an image of a 2. Let's see!

```
import matplotlib.pyplot as plt  # pip install matplotlib

plt.imshow(data[0][0].view(28,28))
plt.show()
```



Clearly a 2 indeed.

So, for our checklist:

We've got our data of various featuresets and their respective classes.
That data is all numerical.
We've shuffled the data.
We've split the data into training and testing groups.
Is the data scaled?
Is the data balanced?

Looks like we have a couple more questions to answer. First off is it scaled? Remember earlier I warned that the neural network likes data to be scaled between 0 and 1 or -1 and 1. Raw imagery data is usually RGB, where each pixel is a tuple of values of 0-255, which is a problem. 0 to 255 is not scaled. How about our dataset here? Is it 0-255? or is it scaled already for us? Let's check out some lines:

```
data[0][0][0][0]
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.])
```

Hmm, it's empty. Makes sense, the first few rows are blank probably in a lot of images. The 2 up above certainly is.

/

```
data[0][0][0][3]
```

```
tensor([0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0118, 0.4157, 0.9059, 0.9961, 0.9216, 0.5647, 0.1882, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000])
```

Ah okay, there we go, we can clearly see that... yep this image data is actually already scaled for us.

... in the real world, it wont be.

Like I said: Cheating! Hah. Alright. One more question: Is the data balanced?

What is data balancing?

Recall before how I explained that if we don't shuffle our data, the machine will learn things like what the last few hundred classes were in a row, and probably just predict that from there on out.

Well, with data balancing, a similar thing could occur.

Imagine you have a dataset of cats and dogs. 7200 images are dogs, and 1800 are cats. This is quite the imbalance. The classifier is highly likely to find out that it can very quickly and easily get to a 72% accuracy by simple always predicting `dog`. It is highly unlikely that the model will recover from something like this.

Other times, the imbalance isn't quite as severe, but still enough to make the model almost always predict a certain way except in the most obvious-to-it-of cases. Anyway, it's best if we can balance the dataset.

By "balance," I mean make sure there are the same number of examples for each classifications in training.

Sometimes, this simply isn't possible. There are ways for us to handle for this with special class weighting for the optimizer to take note of, but, even this doesn't always work. Personally, I've never had success with this in any real world application.

In our case, how might we confirm the balance of data? Well, we just need to iterate over everything and make a count. Pretty simple:

```
total = 0
counter_dict = {0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0, 9:0}


for data in trainset:
    Xs, ys = data
    for y in ys:
        counter_dict[int(y)] += 1
        total += 1

print(counter_dict)

for i in counter_dict:
    print(f"{i}: {counter_dict[i]/total*100.0}%")
```

```
{0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851,
0: 9.871666666666666%
1: 11.236666666666666%
2: 9.93%
3: 10.218333333333334%
4: 9.736666666666666%
5: 9.035%
6: 9.863333333333333%
7: 10.441666666666666%
8: 9.751666666666667%
9: 9.915000000000001%
```

I am sure there's a better way to do this, and there might be a built-in way to do it with `torchvision`. Anyway, as you can see, the lowest percentage is 9% and the highest is just over 11%. This should be just fine. We could balance this perfectly, but there's likely no need for that.

I'd say we're ready for passing it through a neural network, which is what we'll be doing in the next tutorial.

/