PyTorch Layer Dimensions: Get your layers to work every time (the complete guide)

Get your layers to fit smoothly, the first time, every time. A starter's guide to becoming fluent in tensor and layer dimensions in PyTorch.

Jake Krajewski · Follow
Published in Towards Data Science · 8 min read · Jan 11, 2020

Get your layers to fit smoothly, the first time, every time with this invaluable knowledge. — Photo by Clark Van Der Beken on Unsplash

Preface

This article covers defining tensors, and properly initializing neural network layers in PyTorch, and more! (Formerly titled PyTorch layer dimensions: What size and why?)

. . .

Introduction

You might be asking: "How do I initialize my layer dimensions in PyTorch without getting yelled at?" Is it all just trial and error? No, really… What are they supposed to be? For starters, did you know that the *first two* required arguments of a `torch.nn.Conv2d` layer, and a `torch.nn.Linear` layer ask for completely different aspects of the exact same tensor data? If you didn't know this, keep reading.

Example 1: Same, same, but different.

> Constructing a convolution layer and linear layer are syntactically similar, but the args do not expect similar things, despite being able to operate on the exact same input data (although that data should be sized differently).

```
# The __init__ method of a nn.Module class:
...
def __init__(self):
"""Initialize neural net layers."""
    super(Net, self).__init__()

    # Intialize my 2 layers here:

    self.conv = nn.Conv2d(1, 20, 3) # Give me depth of input.

    self.dense = nn.Linear(2048, 10) # Give me features of input.
...
```

You need to develop your understanding of how PyTorch models would like to consume data before just throwing a dataset at some network layers.

. . .

Lesson 1: How to read tensor sizes in PyTorch

Below are some common tensor sizes encountered in PyTorch and typical examples of when to utilize them. It's important to know what you're looking at, because their structures are not as predictable as one might desire (this somewhat unintuitive design choice was implemented primarily for performance benefits, *which is* OK… *I guess*).

One mental anchor we have when feeding tensors into convolutional or linear layers *(although not RNN's)* is that the first dimension is always **batch size (N).** Whereas, the remaining dimensions depend on what phase the moon is in and the intermittent frequency of cricket *(Acheta domesticus)* chirps at moonrise in your region. Just kidding, it's not that simple. You have to learn them by rote. So start *roting (Example 2 below).*

Take the red pill they said. Go deeper they said. What is the 3rd dimension of this tensor supposed to be?!? — Photo by Tim Gouw on Unsplash

It's important to know how PyTorch expects its tensors to be shaped— because you might be perfectly satisfied that your 28 x 28 pixel image shows up as a tensor of torch.Size([28, 28]). Whereas PyTorch on the other hand, thinks you want it to be looking at your 28 batches of 28 feature vectors. Suffice it to say, you're not going to be friends with each other for a little while until you learn how to see things her way — so, don't be that guy. Study your tensor dimensions!

Example 2: The tensor dimensions PyTorch likes.

```
"""Example tensor size outputs, how PyTorch reads them, and where you
encounter them in the wild.

Note: the values below are only examples. Focus on the rank of the
tensor (how many dimensions it has)."""

>>> torch.Size([32])
    # 1d: [batch_size]
    # use for target labels or predictions.

>>> torch.Size([12, 256])
    # 2d: [batch_size, num_features (aka: C * H * W)]
    # use for nn.Linear() input.

>>> torch.Size([10, 1, 2048])
    # 3d: [batch_size, channels, num_features (aka: H * W)]
    # when used as nn.Conv1d() input.
    # (but [seq_len, batch_size, num_features]
    # if feeding an RNN).

>>> torch.Size([16, 3, 28, 28])
    # 4d: [batch_size, channels, height, width]
    # use for nn.Conv2d() input.

>>>  torch.Size([32, 1, 5, 15, 15])
    # 5D: [batch_size, channels, depth, height, width]
    # use for nn.Conv3d() input.
```

> *Notice how the* **Conv2d layer wants a 4d tensor?** *How about the 1d or 3d layers?*

So, if you wanted to load a grey scale, 28 x 28 pixel image into a Conv2d network layer, find the layer type in the example above. Since it wants a 4d tensor, and you already have a 2d tensor with height and width, just add batch_size, and channels *(see rule of thumb for channels below)* to pad out the extra dimensions, like so: [1, 1, 28, 28]. That way, you and PyTorch can make up and be friends again.

. . .

Lesson 2: Initializing a torch.nn.Conv2d layer

```
"""
Class

torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1,
padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')

Parameters

in_channels (int) – Number of channels in the input image
out_channels (int) – Number of channels produced by the convolution
"""
```

Remember which dimension your input channels are? See *Lesson 1* if you forgot. Simply use that number for your `in_channels` argument in the first convolutional layer. *Done.*

> ***Rule of thumb for "in_channels" on your first Conv2d layer:***
>
> *— If your image is black and white, it is 1 channel. (You can ensure this by running `transforms.Grayscale(1)` in the transforms argument of the dataloader.)*
>
> *— If your image is color, it is 3 channels (RGB).*
>
> *— If there is an alpha (transparency) channel, it has 4 channels.*

This means for your first Conv2d layer, even if your image size is something enormous like 1080px by 1080px, your `in_channels` will typically be either 1 or 3.

Note: *If you tested this with some randomly generated tensor and it throws up at you still and you're yelling at your computer right now, breathe. It's OK. Make sure it has the right dimensions. Did you `unsqueeze()` the tensor? Pytorch wants batches. The unsqueeze() function will add a dimension of 1 representing a batch size of 1.*

But, what about out_channels?

What about the `out_channels` you say? That's *your choice* for how deep you want your network to be. Basically, your `out_channels` dimension, defined by Pytorch is:

> ***out_channels*** *(int) — Number of channels produced by the convolution*

For each convolutional kernel you use, your output tensor becomes one channel deeper when passing through that layer. If you want a ton of kernels, make this number high like 121, if you want just a few, make this number low like 8 or 12. Whatever number you choose here will be the value for `channels_in` of the ***next convolutional*** layer, and so on and so forth.

> *Note: The value of `kernel_size` is custom, and although important, doesn't lead to head-scratching errors, so it is omitted from this tutorial. Just make it an odd number, typically between 3–11, but sizes may vary between your applications.*

Generally, convolutional layers at the front half of a network get deeper and deeper, while fully-connected (aka: linear, or dense) layers at the end of a network get smaller and smaller. Here's a valid example from the 60-minute-beginner-blitz (notice the *out_channel* of `self.conv1` becomes the *in_channel* of `self.conv2`):

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square
convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)

        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image
dimension
```

```
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

Let's talk about fully connected layers now.

. . .

Lesson 3: Fully connected (torch.nn.Linear) layers

Documentation for Linear layers tells us the following:

```
"""
Class

torch.nn.Linear(in_features, out_features, bias=True)

Parameters

in_features – size of each input sample
out_features – size of each output sample

"""
```

I know these look similar, but do not be confused: "`in_features`" and "`in_channels`" are completely different, beginners often mix them up and think they're the same attribute.

```
# Asks for in_channels, out_channels, kernel_size, etc
self.conv1 = nn.Conv2d(1, 20, 3)

# Asks for in_features, out_features
self.fc1 = nn.Linear(2048, 10)
```

Calculate the dimensions.

There are two, specifically important arguments for all `nn.Linear` layer networks that you should be aware of no matter how many layers deep your network is. The **very first argument**, and the **very last argument**. It doesn't matter how many fully connected layers you have in between, those dimensions are easy, as you'll soon see.

If you want to pass in your 28 x 28 image into a linear layer, you have to know two things:

1. **Your 28 x 28 pixel image can't be input as a [28, 28] tensor.** This is because `nn.Linear` will read it as 28 batches of 28-feature-length vectors. Since it expects an input of `[batch_size, num_features]`, you have to transpose it somehow (see *view()* below).
2. **Your batch size passes unchanged through all your layers.** No matter how your data changes as it passes through a network, your first dimension will end up being your `batch_size` even if you never see that number explicitly written anywhere in your network module's definition.

> ***Use view() to change your tensor's dimensions.***
>
> `image = image.view(batch_size, -1)`
>
> *You supply your batch_size as the first number, and then "-1" basically tells Pytorch, "you figure out this other number for me… please." Your tensor will*

> *now feed properly into any linear layer. Now we're talking!*

So then, to initialize the **very first argument** of your linear layer, pass it the number of features of your input data. For 28 x 28, our new view tensor is of size [1, 784] (1 * 28 * 28):

Example 3: Resize with view() to fit into a linear layer

```
batch_size = 1

# Simulate a 28 x 28 pixel, grayscale "image"
input = torch.randn(1, 28, 28)

# Use view() to get [batch_size, num_features].
# -1 calculates the missing value given the other dim.
input = input.view(batch_size, -1) # torch.Size([1, 784])

# Intialize the linear layer.
fc = torch.nn.Linear(784, 10)

# Pass in the simulated image to the layer.
output = fc(input)

print(output.shape)
>>> torch.Size([1, 10])
```

> *Remember this — if you're ever transitioning from a convolutional layer output to a linear layer input, you must resize it from 4d to 2d using view, as described with image example above.*
>
> *So, a conv output of [32, 21, 50, 50] should be "flattened" to become a [32, 21 * 50 * 50] tensor. And the in_features of the linear layer should also be set to [21 * 50 * 50].*

The **second argument** of a linear layer, if you're passing it on to more layers, is called **H** for hidden layer. You just kind of play positional ping-pong with H and make it the last of the previous and the first of the next, like this:

```
"""The in-between dimensions are the hidden layer dimensions, you
just pass in the last of the previous as the first of the next."""

fc1 = torch.nn.Linear(784, 100) # 100 is last.
fc2 = torch.nn.Linear(100, 50) # 100 is first, 50 is last.
fc3 = torch.nn.Linear(50, 20) # 50 is first, 20 is last.
fc4 = torch.nn.Linear(20, 10) # 20 is first.

"""This is the same pattern for convolutional layers as well, only
it's channels, and not features that get passed along."""
```

The **very last output**, aka your **output layer** depends on your model and your loss function. If you have 10 classes like in MNIST, and you're doing a classification problem, you want all of your network architecture to eventually consolidate into those final 10 units so that you can determine which of those 10 classes your input is predicting.

The last layer is dependent on what you want to infer from your data. The operations you can do to get the answer you need is a topic for another article, because there is a lot to cover. But for now you should have all the basics covered.

. . .

That's it!

You should now be able to build a network without scratching your head or getting shouted at by the interpreter. Remember, your batch_size or dim 0 is the same, all the way through. Your convolution layers care about depth (channels), and your linear layers care about feature counts. And learn how to read those tensors!

Please leave a comment, or share this article if you liked it and found it helpful!

Machine Learning Deep Learning Pytorch Dimensions Neural Network

Written by Jake Krajewski

Follow

226 Followers · Writer for Towards Data Science

Cognitive Science Master, Experienced digital product designer. Formerly @GoPro . Exploring the intersection of tech, startups, and a.i./deep learning

New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 4

1.7K 17

Display Anything with E-Ink by Making an E-Paper Device.

Difficulty: Advanced. Step-by-step instructions on how to connect an e-ink...

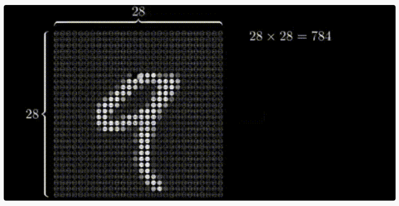
· 11 min read · Sep 3, 2020

178 5

See all from Jake Krajewski

See all from Towards Data Science

Recommended from Medium



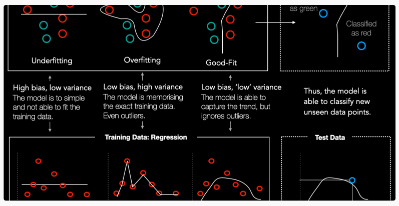
Sadaf Saleem

Neural Networks in 10mins. Simply Explained!

What are Neural Networks?

9 min read · May 15

262 2



Frederik vl in Advanced Deep Learning

Understanding Bias and Variance in Machine Learning

The terms bias and variance describe how well the model fits the actual unknown data...

3 min read · Sep 16

45

Lists

Predictive Modeling w/ Python

20 stories · 457 saves

Practical Guides to Machine Learning

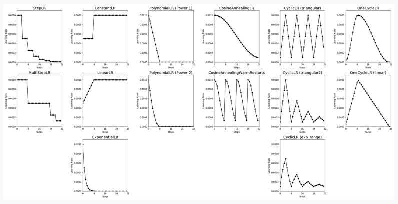
10 stories · 529 saves

Natural Language Processing

674 stories · 293 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 134 saves



Leonie Monigatti in Towards Data Science

A Visual Guide to Learning Rate Schedulers in PyTorch

LR decay and annealing strategies for Deep Learning in Python

· 9 min read · Dec 7, 2022

xxorxxo

Basics for PyTorch

PyTorch's Basic Syntax

2 min read · Sep 11

1.1K 7

4

Anthony Peng in Polo Club of Data Science

Multi-GPU Training in PyTorch with Code (Part 3): Distributed Data...

We discussed single-GPU training in Part 1 and multi-GPU training with DP in Part 2. In...

12 min read · Jul 7

11

Rayyan Shaikh

Mastering BERT: A Comprehensive Guide from Beginner to Advanced...

Introduction: A Guide to Unlocking BERT: From Beginner to Expert

19 min read · Aug 26

1.5K 12

See more recommendations