Blue letters are code, Green is comments, Orange is code output, Purple and black are just words. Also, this is a general intro to Pytorch. A lot of stuff could’ve been left out for the purposes of chess, but I wanted it to be more holistic(like dataset and data transfer stuff).

What's a Pytorch: Pytorch contains math and array-like functionality(in PT they are called tensors) that is very much like numpy, but GPU optimized. Also, it contains a library with cool building blocks for neural nets and functionality for automatically computing derivatives for backprob.

Things:

1. Tensors and CUDA
2. Data Management(not needed for chess)
3. How to build a NN
4. Autograd, backward passes, and loss functions
5. Training, optimization, and checkpoints
6. Evaluating a model
7. Example full code structure
8. Pretrained models

1: Tensors and CUDA

Tensors in Pytorch are like arrays in numpy! Indexing and slicing is identical. You can also initialize them similarly:

1. From data

    ```
    data = [[1, 2],[3, 4]]
    x_data = torch.tensor(data)
    ```

    Or

    ```
    x=torch.tensor([1,2,3])
    ```

2. From np array

    ```
    np_array = np.array(data)
    x_np = torch.from_numpy(np_array)
    ```

3. From tensor

    ```
    x_ones = torch.ones_like(x_data) #same shape and datatype
    x_rand = torch.rand_like(x_data, dtype=torch.float) #same shape new datatype
    ```

4. With values

    ```
    shape = (2,3,)
    rand_tensor = torch.rand(shape)
    ones_tensor = torch.ones(shape)
    zeros_tensor = torch.zeros(shape)
    empty_tensor = torch.empty(shape)
    ```

Just like numpy, torch comes with a lot of functions built in for manipulating tensors. If you need something, it probably exists. A few common examples are (add, cat, mul, matmul, ones, view).

View’s name is a little misleading. It is the function for reshaping:

```
y = torch.rand([2,5])
print(y)
```

```
tensor([[0.2632, 0.9136, 0.5702, 0.9915, 0.4885],
        [0.5025, 0.7631, 0.5265, 0.4594, 0.1881]])
```

```
y.view([1,10])
```

```
tensor([[0.2632, 0.9136, 0.5702, 0.9915, 0.4885, 0.5025, 0.7631, 0.5265, 0.4594,
         0.1881]])
```

Tensors have three attributes, two mirror np, one is unique to tensors and why we use them:

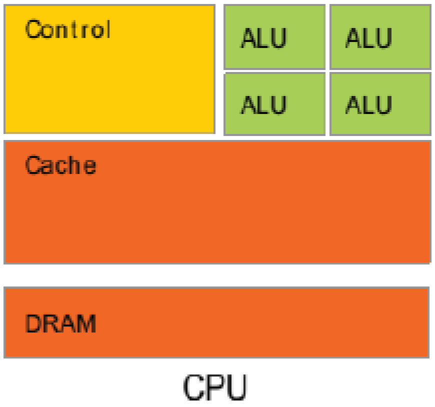
1. tensor.shape
2. tensor.dytpe
3. tensor.device

Shape: A tuple describing (rows, cols, depth, more crazy dims, etc…)
Dtype: Type of data in the tensor (float, int, etc…)
Device: This is the cool one! Pytorch can store Tensors on the GPU for muuuuuuch faster operations(especially in a nn setting)

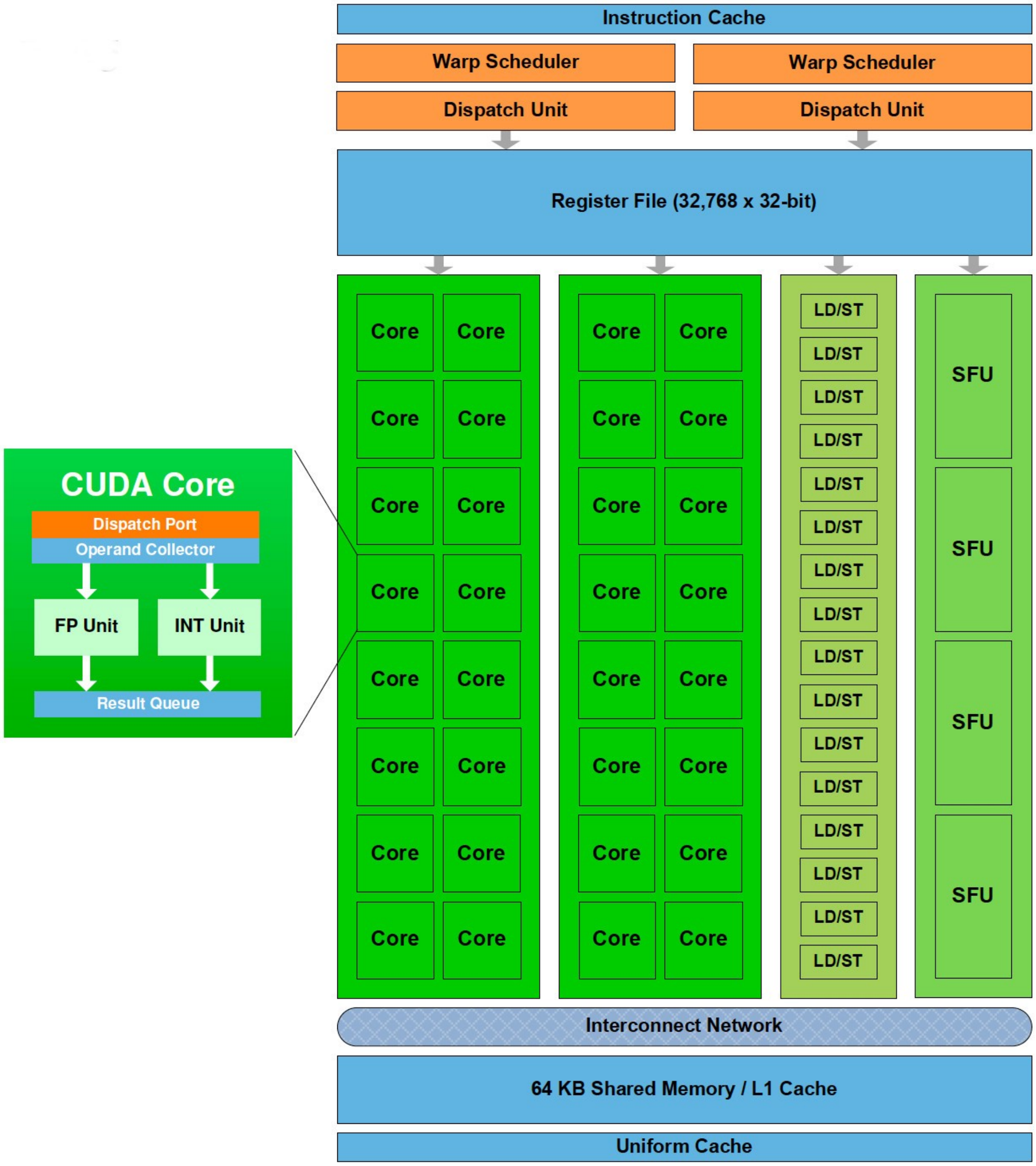
But why are GPUs better? It seems like most things run just dandy with a CPU:
The main design difference between CPUs and GPUs is latency vs throughput. CPUs have a small amount of very capable cores with the idea that any instruction can be handled efficiently. They are able to interpret a wide variety of instructions and any one instruction is completed very quickly(low latency). GPUs on the other hand have maaaaany dumb cores that are specifically good at floating point operations. This makes them best for doing repetitive bulk operations; while they may not get any one operation done as fast as a CPU, they will be much faster if that operation needs to be done hundreds of times in parallel(like pixels in video rendering, matrix mults, etc). In this sense, they optimize throughput over latency: they want to get as much net stuff done without caring as much about how fast each individual thing is.

This notion is reflected in CPU hardware as well. CPUs reserve large portions of their cores for control logic, arithmetic logic units(ALU), and register/cache memory. CPUs take this approach because they expect many types of instructions(read, write, branch, jump, arithmetic) and want each individual instruction type to have low latency. Unlike the GPU, the CPU devotes lots of space to a cache block, which is small, fast access memory(for things like saving variables and such) and the control block, which is able to decode many instruction types and contains hardware like a branch predictor* to speed up branching. A CPU has one to dozens of cores, and a super high level image of a core is shown below:

*The branch predictor makes a guess where something like an if statement will lead in code. Then, it continues to load instructions based on its guess until the core evaluates whether the guess was right. If it was, the next instructions are loaded already and CPU cycles are saved, otherwise, it uses its control unit to discard the preloaded instructions and takes the correct branch route. This is especially useful for things like while loops, where one outcome of the branch(looping) is very likely and can keep being predicted until the loop ends.



The GPU hardware is a little different. Much less space is devoted to local cache and control instructions, and much more space is devoted to CUDA cores(if it's NVIDIA, other companies have a similar thing with a different name). These CUDA cores contain a unit for float operations, int operations, or both. Also, SFUs(special function units) are available for more complex operations. Overall, a much greater proportion of the surface area is devoted to individual computational units. This is the case because the GPU doesn’t expect to do memory reads and writes, complex control logic, or really anything that isn’t parallel math. These CUDA cores are grouped by 16 or 32 into a streaming multiprocessor(shown below):



Fermi Streaming Multiprocessor (SM)

In practice, the programmer defines tasks and the GPU allocates them to SMs with no guarantee of when or where they will be completed. It doesn’t wait for slow tasks because it is maximizing throughput, not latency.

But what about CPU parallelism? My CPU can do a lot with its fast cores right?
What a good question, that's true! A modern consumer CPU has a few ways to be parallel. For example, vectorized code uses Single Instruction Multiple Data to do multiple computations within one instruction. Vectorized code works because registers are larger than many inputs. Intel’s current instruction set uses 256-bit registers. When vectorized code is written, these can

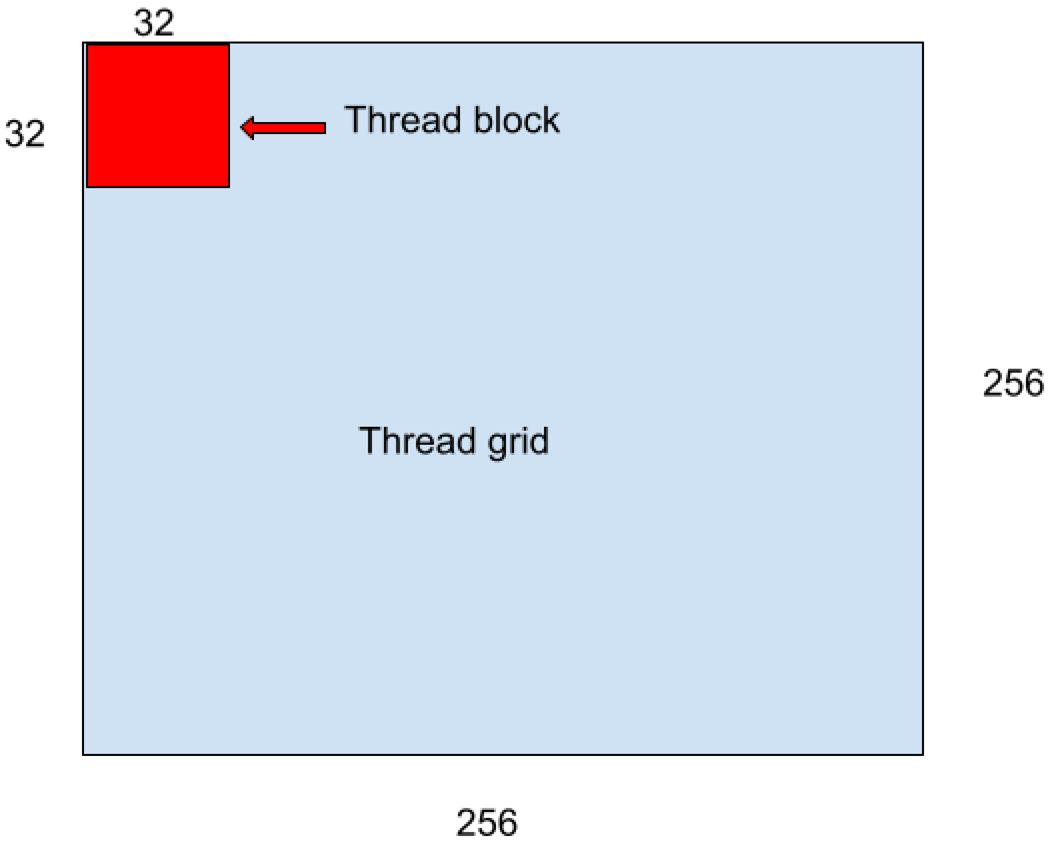
hold(and operate on) more than one operand apiece. For example, if you wrote vectorized code adding one to a matrix of 32 bit ints, each register could hold and work on 8 ints at once!

Also, there is often instruction-level parallelism. In a simple instruction like adding two integers, there are a few steps: loading the instruction and increasing the program counter to point to the next instruction(using control), decode the instruction and read the source registers(using registers to read numbers), execute the instruction(add using ALU), and write the new number to memory(using Dram or cache). Since all these steps use different parts of the CPU, we can overlap processes. With consecutive add instructions, one can be loading numbers with registers while adding with the ALU, or decoding with the controller while writing to DRAM and so on. In general, this kind of parallelism allows for about 2 instructions per cycle when varied instructions are used.

Finally, a CPU has multiple cores. A modern consumer CPU has like 8 cores for added parallelism. Taking all this into account, a CPU can work with 8 cores, 2 instructions at a time, and 8 operands at once for 128-way parallelism! While this sounds good, it's really not impressive. A modern GPU can launch tens of thousands of threads at once. Even though the GPU clock is slower than the CPU, the sheer amount of parallelism results in waaay more throughput for repetitive, mathematical operations.

This part to the next bold is just a tangent i got lost on. It's a slight intro to how CUDA ties into the hardware. The original title was: but how would you program a GPU?
From an NVIDIA GPU programmer's perspective, you have some abstraction when running something on the GPU. The programmer thinks in terms of threads which are like paths of execution. Each thread has a small, fast local memory. These threads are grouped into thread blocks that all carry out the same task and have a fast shared memory. Blocks are what the GPU allocates to SMs; an SM can run more than one block, but one block must be on the same SM, so there is a maximum block size(usually 1024). Finally, blocks can be grouped into grids. Every thread also has access to a slow global memory and knows its thread indexes, the block size, block index, and the grid size.

To give a more tangible example, consider an operation where the programmer wants to add one to the intensity of each pixel in a 256*256 array of pixels. In this case, each pixel can be thought of as an execution thread. The programmer can also split the 256*256 screen into a grid of 64 32*32 thread blocks(so that each block is 1024 threads). This scenario is shown below:



The overall index of any one pixel can be determined from the gridDim, blockDim, blockIdx, threadIdx.

She would want to write a program that initializes this setup and adds one to each thread in parallel. There's a bunch of memory allocation and such that goes into making a full program, but the main GPU specific code of using a kernel is done with a kernel as follows:

How to Launch a Kernel:

```
Kernel<<<dim 3 grid of blocks, dim3 block of threads>>>(kernel parameters);
```

For this example, let intensityImage be the original and adjustedImage be the new one:

```
Int numToAdd = 1
Kernel<<<dim3(8,8), dim3(32,32)>>>(numToAdd, intensityImage, adjustedImage);
```

And an example Kernel for this problem:

The most difficult part is

```
__global__ void Kernel(int numToAdd,const intensityImage, const adjustedImage) {
int column = threadIdx.x + blockIdx.x*blockDim.x;
int row = threadIdx.y + blockIdx.y*blockDim.y;
int numCols = gridDim.x * blockDim.x;
int threadId = column + row*numCols;
adjustedImage[threadId] = intensityImage[threadId] +1;
}
```

But, how do we specify CPU vs GPU tensors?
We use torch.cuda! Unfortunately this means your machine needs to have CUDA and Cudnn installed on an NVIDIA GPU, which is a biiit of a process. If your machine of choice doesn't have an NVIDIA card, there are a few good alternatives:

Long TLDR: AWS and co gives you any amount of compute power at any time, but only if you have any amount of money. A developer account is $29/month and that usually allows you to do what you need without any extra charges(https://aws.amazon.com/premiumsupport/plans/). Kaggle and Google collab give you a free card in the cloud! The cards are pretty dang good(500+ dollar GPUs), but they make their service annoying to use. Both kick you out for a long idle period and cap the execution time of your code. So, you need to partially train, save weights, and resume often. Kaggle gives a slightly better card usually but Google Colab is easier to setup IMO and gives muuuch more storage space(360 vs 5 GB) for data and such.

1. AWS, Google Cloud, etc..- really good cards but pricey
2. Kaggle notebooks:

Specs:
9 hours of execution time
60 min idle time
5 Gigabytes of auto-saved disk space (/kaggle/working)
16 Gigabytes of temporary, scratchpad disk space (outside /kaggle/working)

CPU specifications
4 CPU cores
16 Gigabytes of RAM

GPU Specifications
2 CPU cores
13 Gigabytes of RAM

3. Google Collab:

Specs:
12 hours of execution time
90 min idle time
360 Gigabytes of auto-saved disk space

CPU specifications
2 CPU cores
12 Gigabytes of RAM

GPU Specifications
2 CPU cores
12 or 16 GB RAM depending on the card you're assigned

Back to the main stuff

Here’s a few key things about interacting with CUDA:

First off, to check if CUDA is ready:

```
torch.cuda.is_available() → bool #returns true or false
```

torch.cuda needs to know exactly where to put tensors. Since systems can have multiple GPUs, it will default to GPU 0 on your device(run nvidia-smi to see your GPU indexes), but functions can specify a different GPU. To see the current GPU(that should default to 0):

```
torch.cuda.current_device() → int
Or
device
```

The returned int is GPU index. To change the default device and see how many devices are available:

```
device = torch.device(“cuda:device_index”) #set device
torch.cuda.device_count() → int   #returns GPU number
```

You can dynamically set the device to allow the code to run on a CPU. This is useful for training small nets like basic classifiers or for running low FPS inference that can work on the CPU:

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")  # you can continue going on here, like cuda:1 cuda:2....etc.
    print("Running on the GPU")
else:
    device = torch.device("cpu")
    print("Running on the CPU")
```

To transfer a thing to the GPU, you can use the .to(device) function. This is super powerful. You can transfer a neural net, a batch of data, a tensor, etc… Keep in mind though that transfers are relatively slow, but they are often unavoidable(like if you can’t store your whole dataset in GPU VRAM you have to pass data in batches).

Neural net example:

```
net.to(device) #move an initialized net called net to the GPU(slow with a big nn)
or
net = Net_class().to(device) #initialize a net with class definition Net_class() directly on the GPU
```

Data batch example:

```
batch = train_data[i:i+BATCH_SIZE]
batch = batch.to(device)
```

Tensor example:

```
tensor = tensor.to(device) #transfer tensor to device
tensor = torch.tensor(torch.rand(1,3), device=torch.device('cuda')) #initialize tensor on GPU(faster but takes VRAM)
```

There are a whole lot of crazy things you can do with torch.cuda but this is a good start(1). The best way to see how this basic CPU/GPU interaction works is to look at and understand the classifier CNN in section 3.

(1) An example of a crazy thing you can do is cuda streams. These are like CPU threads, where you can explicitly define sequences of operations and run them on GPUs. This increases parallelization and can improve program runtime! However, since everything is parallelized, you have to be careful about timings and synchronization. For example, if one stream calculated a sum and another stream depended on the sum, you would have to explicitly make sure the calculation occurred on first with synchronize statements. Luckily, pytorch creates one default stream per GPU, and takes care of timing on that stream!

2. Data Management(not needed for chess)

Loading Pytorch datasets:

A key aspect of Pytorch is its inclusion of many common datasets with “torchvision”. The list of 27 datasets is here(https://pytorch.org/docs/stable/torchvision/datasets.html). Here is an example of loading the MNIST dataset from the datasets library:

```
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()
                    ]))
```

There is a slightly different process for getting the train and testing sets from the datasets library. The different processes for each dataset are also on this page: (https://pytorch.org/docs/stable/torchvision/datasets.html).

Defining custom datasets in Pytorch:
A common approach is to create a class for parsing your folder directories for data. This is done with a Python Dataset class. To create a dataset in Pytorch, you should inherit torch.utils.data.Dataset and you muuuuust implement an __init__, __len__, and __getitem__ for the dataset. Init is the class initialization function, len returns the Dataset length, and getitem returns a specific index. You can also add new functions as necessary.

For an example:

```
From torch.utils.data import Dataset

Class numbersDataset(Dataset):
    def __init__(self, low, high):
        self.samples = list(range(low,high))
    def __len__(self):
        return len(self.samples)
    def __getitem__(self, idx):
        return self.samples[idx]

if __name__ ==’__main__’:
    dataset = numbersDataset(0, 101) #make the dataset the list of numbers from 0 to 100
```

Here, we make a class defining the dataset and then create the dataset from that class in the main function. In this instance, our Dataset is just the list of numbers from a low number to a high number. With the dataset properly initialized, things like the following work:

```
print(len(dataset))
print(dataset[10]))
print(dataset[10:15])
```

We can also extend this to load a dataset from a folder directory and text files. For this example, we have the following folder structure:

```
students
    freshman
        CS
            Natalia OM
            etc…
            Ari Conati
        etc…
        Engineering
            Marco Conati
            etc…
```

```
                Darko Monati
    etc...
    senior
        CS
                Big Nut
                etc…
                Aery Conaeti
        etc…
        Engineering
                Puddle
                etc…
```

To parse this directory and create a dataset of all students:

```
import os
from torch.utils.data import Dataset

class studentsDataset(Dataset):
    def __init__(self, data_root):
        self.samples = []

        for classYear in os.listdir(data_root):
            classYear_folder = os.path.join(data_root, classYear)

            for major in os.listdir(classYear_folder):
                major_filepath = os.path.join(classYear_folder, major)

                with open(major_filepath, 'r') as major_file:
                    for name in major_file.read().splitlines():
                        self.samples.append((classYear, major, name))

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        return self.samples[idx]


if __name__ == '__main__':
    dataset = studentsDataset('/home/marco/Data/students/')
    print(len(dataset))
    print(dataset[420])
```

Here, we initialize the dataset by iteratively searching each folder in the file structure within the __init__ function. Examples of what could be printed by the print statements are:

```
500
(‘senior’, ‘Big Nut’, ‘CS’)
```

Loading custom image datasets:

To load custom image datasets, you can also use the torchvision.datasets.ImageFolder function(also on https://pytorch.org/docs/stable/torchvision/datasets.html). To set up the image folders, its easiest to set the up in the following kind of folder structure:

```
Images
        train
                Class 1
                        imInClass1.png
                        etc...
                        wowThisIsSoCool.png
                etc...

                Class n
                        whatAGoodGuide.png
                        etc…
                        imSoImpressed.png
        Test
                Class 1
                        newClass1Image.png
                        etc...
                        zoinksThisIsAlsoCool.png
                etc...

                Class n
                        sorryImGettingSleepy.png
                        etc…
                        iHopeThisAllMakesSense.png
```

Then, you can load the data like this:

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

TRAIN_DATA_PATH = "./images/train/"
TEST_DATA_PATH = "./images/test/"
```

```
train_data = torchvision.datasets.ImageFolder(root=TRAIN_DATA_PATH)
test_data = torchvision.datasets.ImageFolder(root=TEST_DATA_PATH)
```

Using a dataloader and transforms:

There are a couple things you might want to do differently from the approach above. First off, the above method is loading the entire dataset at once. You could load batches from this data manually, but it is a bit difficult to implement efficiently.

```
DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
        batch_sampler=None, num_workers=0, collate_fn=None,
        pin_memory=False, drop_last=False, timeout=0,
        worker_init_fn=None, *, prefetch_factor=2,
        persistent_workers=False)
```

Where:

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: 1).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: False).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any Iterable with __len__ implemented. If specified, shuffle must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
- **collate_fn** (*callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – set to True to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If False and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: False)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
- **worker_init_fn** (*callable, optional*) – If not None, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: None)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of sample loaded in advance by each worker. 2 means there will be a total of 2 * num_workers samples prefetched across all workers. (default: 2)

- **persistent_workers** (*bool*, *optional*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive. (default: False)

Honestly, I'm not sure what a lot of the parameters do, and how useful they are. The main ones to consider are dataset, batch_size, shuffle, and num_workers.
Dataset: The dataset needs to point to a loaded dataset.
Batch_size: batch size is the amount of images or samples trained on at once. Generally, it is a multiple of 2 and usually around 32. Larger batch sizes give gradient updates based on more information, so they are generally better. However, unless you have a good GPU you can't fit a ton of high res images in VRAM, so I've had to train with small batch sizes before(8).
shuffle: If this is set to true, you will draw a random sample each epoch.
num_workers: workers are CPU based processes that preload batches so that when the training process needs them, they are ready to be moved to the GPU. When it is 0, data is loaded by the main process when each training epoch finishes(this is a bottleneck where training stops to wait for data). A higher number of workers means that multiple batches can be preloaded so that there is no waiting for a batch before the CPU-->GPU transfer. However, these workers also eat up RAM on the CPU(and if memory pin is true they will also eat up GPU VRAM). You can experiment with this parameter, slowly increasing it and looking at memory and speed. People say good starting points are num_workers = 4*num_GPUs or num_workers = num_CPU cores from their experience.

To access data from a dataloader, the following iteration structure is common:

```
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

for epoch in range(EPOCH):
   for batch_index, (x, y) in enumerate(train_loader):   # gives batch data
```

Here, a dataloader is initialized. Then, at each epoch the dataloader iterates over the entire dataset returning x and y corresponding to an input and label along with the batch_index. You can do a similar thing a bunch of ways. One useful note for debugging is that you can create an iterator for the dataloader and iterate it one step at a time manually:

```
train_loader = Data.DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)
it = iter(train_loader)
first = next(it)
second = next(it)
```

You can also create a custom dataloader if you're way smarter than me haha.

Transforms: It's also super common to preprocess data to resize images, randomly flip them, normalize them, etc. This is easy to do when initializing a dataset with datasets.ImageFolder:

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

TRAIN_DATA_PATH = "./images/train/"
TEST_DATA_PATH = "./images/test/"

TRANSFORM_IMG = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(256),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225] )
    ])

train_data = torchvision.datasets.ImageFolder(root=TRAIN_DATA_PATH,
transform=TRANSFORM_IMG)
test_data = torchvision.datasets.ImageFolder(root=TEST_DATA_PATH,
transform=TRANSFORM_IMG)
test_data_loader  = data.DataLoader(test_data, batch_size=BATCH_SIZE, shuffle=True,
num_workers=4)
train_data_loader = data.DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True,
num_workers=4)
```

Here, we define a transform and apply it to our data when we create our train_data and test_data sets. We also create dataloaders because they are cool. Torchvision comes with all kinds of crazy transforms(https://pytorch.org/docs/stable/torchvision/transforms.html). Doing things like randomly flipping, cropping, and rotating images can be super useful to augment a dataset and make it both more general and bigger.

3. How to build a NN

Initialize a NN and its layers:

Now we can get to how we lay out a neural net in Pytorch. It is surprisingly easy given the torch.nn library! In short, we just create a class for our neural net, define the layers that are used, and initialize the model.

To get started, we should import the relevant things from PT:

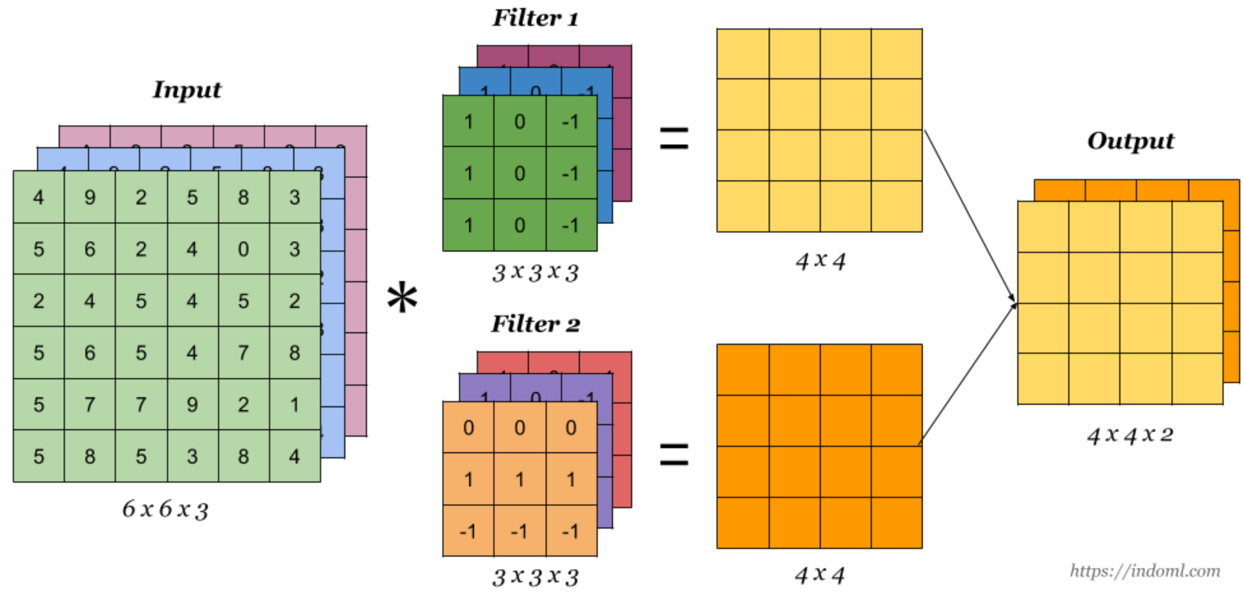
```
import torch.nn as nn
import torch.nn.functional as F
```

Torch.nn and torch.nn.functional are really similar. They contain all the same types of functions(layers, activation functions, etc.). In general, it seems like you should use nn for all layers. This is especially true for layers with learnable parameters as the parameters must be in the __init__ function. nn.functional is generally used for simple operations like \ and softmax. This is the beeeeest explanation I could find online(https://discuss.pytorch.org/t/beginner-should-relu-sigmoid-be-called-in-the-init-method/18689/6), but it's not exactly a rock solid source.

Let's look at what layers are included in torch.nn. The most common types of layers are listed below.

```
torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1, padding: Union[T, Tuple[T, T]] = 0, dilation: Union[T, Tuple[T, T]] = 1, groups: int = 1, bias: bool = True, padding_mode: str = 'zeros')
nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
```

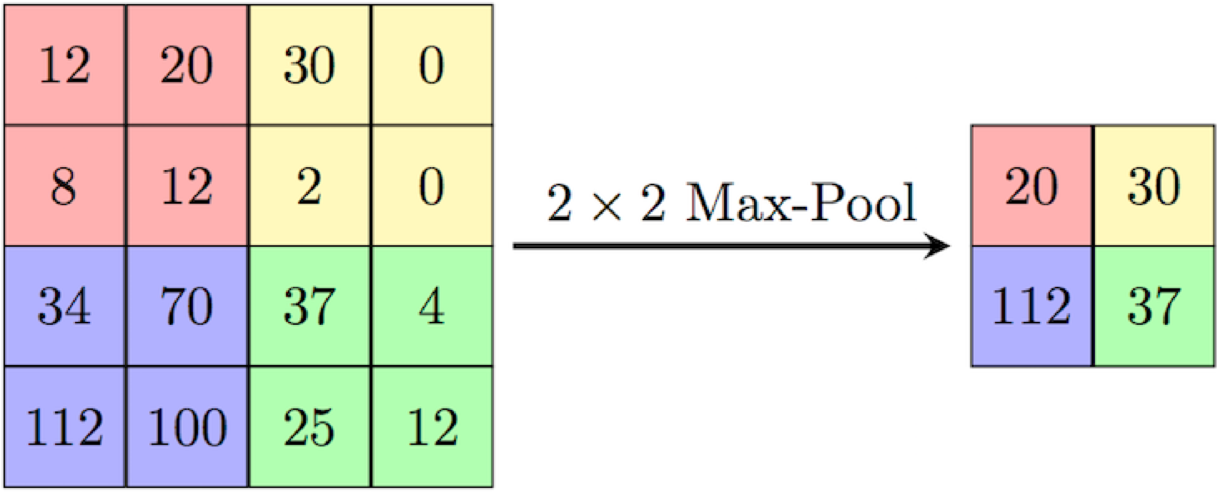
Usage: Takes the 2D convolution of an input tensor(This is the most common type of conv for images/computer vision).



Important Parameters: in_channels-'depth' dimension of the input tensor(3 for an RGB image), out_channels-number of filters in the convolve operation, Kernel_size- width and height of the filters, stride-how far the kernel is slid in the convolution, padding-numbers added to the perimeter of the input pre convolution, padding_mode-type of padding used(zeros, constant numbers, mirror)
Example: self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride = 2)

torch.nn.MaxPool2d(*kernel_size: Union[T, Tuple[T, ...]], stride: Optional[Union[T, Tuple[T, ...]]] = None, padding: Union[T, Tuple[T, ...]] = 0, dilation: Union[T, Tuple[T, ...]] = 1, return_indices: bool = False, ceil_mode: bool = False*)

Usage: Takes the maximum value from each kernel and downsamples the input tensor. The example below has kernel_size 2 and stride 2 with no padding.



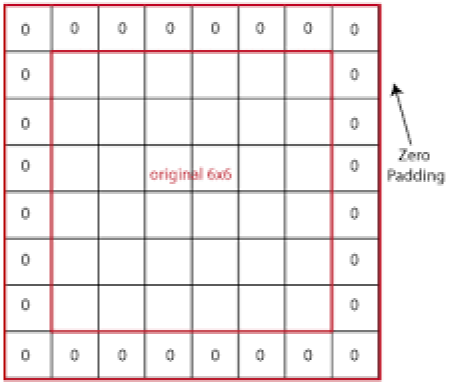
12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

20	30
112	37

Important Parameters: Kernel_size- height and width of the kernel for pooling, stride-how far the kernel moves between each pooling operation, padding- numbers added to the perimeter of the input pre-pooling
Example: self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
Similar options: Average pool

torch.nn.ZeroPad2d(*padding: Union[T, Tuple[T, T, T, T]]*)

Usage: A layer specifically meant for padding the input kernel

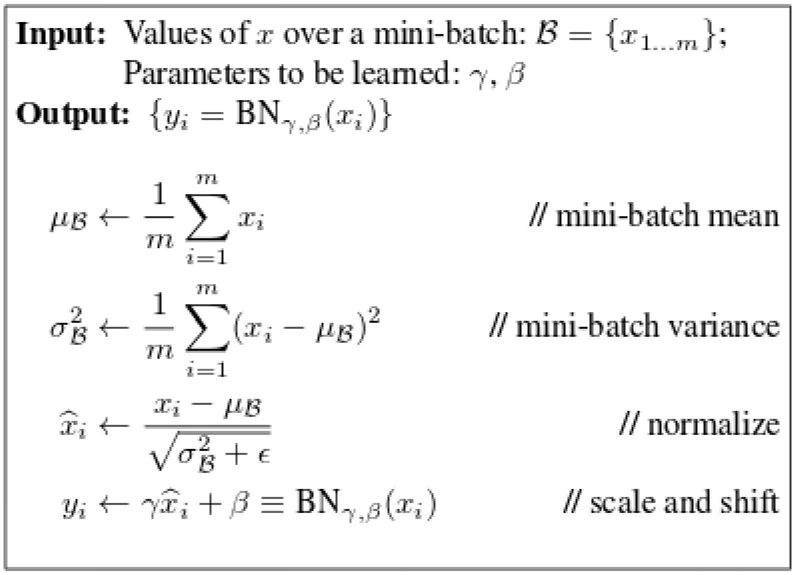


final 8x8

Example: torch.nn.ZeroPad2d(1)
Similar options: Reflection pad and constant pad

torch.nn.BatchNorm1d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Usage: Normalizes layers within the nn on batches of data to promote stable gradients and faster training. This is meant for 4d inputs of dimension(NxCxL) where N is the batch size, C is the channel number, and L is the length of the input.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Important Parameters: num_features-number of input channels to the batch norm

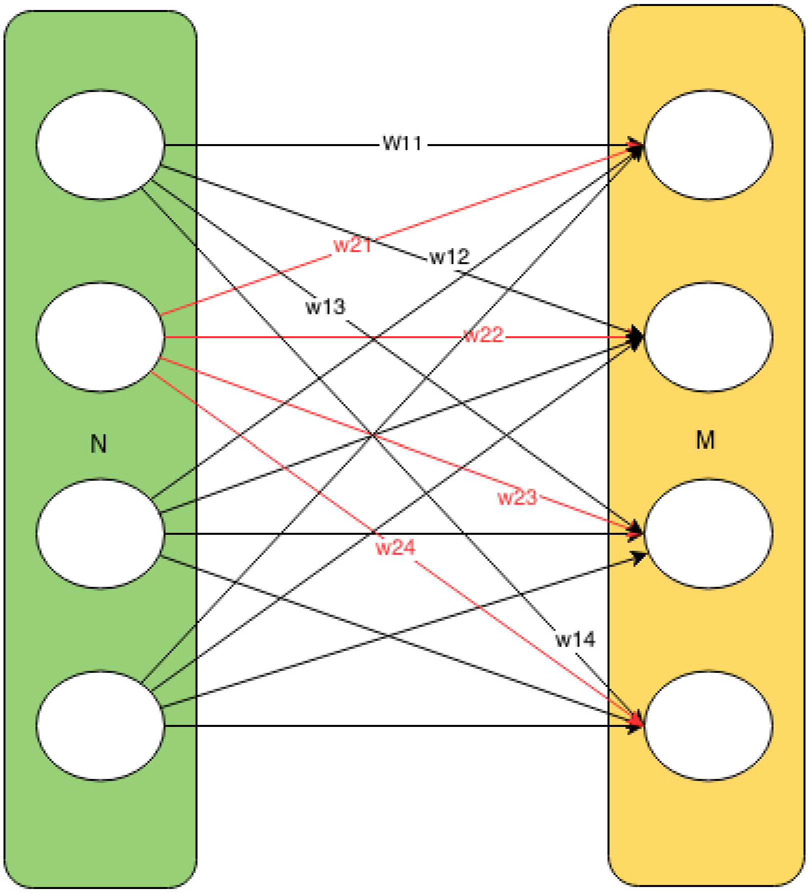
Example: Performs batch norm on data with length 120 nn.BatchNorm1d(120)

torch.nn.BatchNorm2d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Usage: Normalizes batches within the nn to promote stable gradients and faster training. This is meant for 4d inputs of dimension(NxCxHxW) where N is the batch size, C is the channel number, and W H and height and width.
Example: Performs batch norm on data with 6 channels
nn.BatchNorm2d(6)

torch.nn.Linear(*in_features: int*, *out_features: int*, *bias: bool = True*)

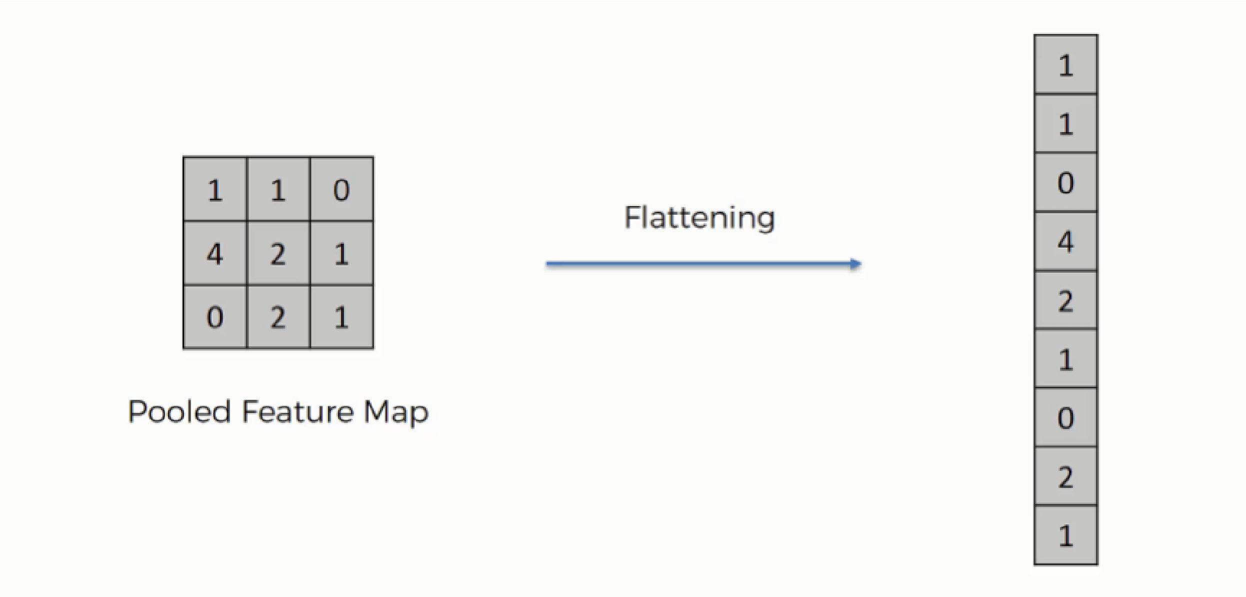
Usage: Implements a fully connected layer



Important Parameters: in_features-number of nodes in input, out_features- number of nodes in the output
Example: nn.Linear(in_features=60, out_features=10)

torch.nn.Flatten(*start_dim: int = 1*, *end_dim: int = -1*)

Usage: Flattens a tensor to reduce dimensionality



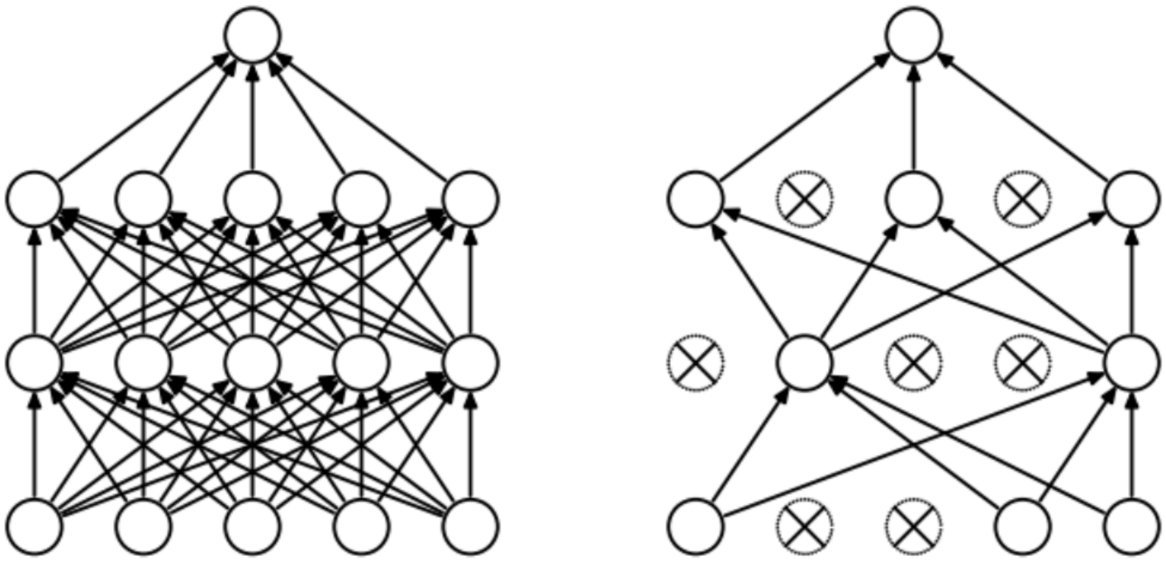
1	1	0
4	2	1
0	2	1

1
1
0
4
2
1
0
2
1

Important Parameters: start_dim-first dimension to flatten, end_dim-last dim to flatten
Example: nn.Flatten(start_dim=1)
Similar options: Reshaping the data with torch.view

torch.nn.Dropout(*p: float = 0.5*, *inplace: bool = False*)

Usage: Randomly zeroes out neurons. This is done to reduce overfitting



(a) Standard Neural Net (b) After applying dropout.

Important Parameters: p- probability to zero out each neuron
Example: torch.nn.Dropout(0.2), #20% probability

torch.nn.funtional docs are here(https://pytorch.org/docs/stable/nn.functional.html). As I tried to explain above, usually functional is just used for basic operations. Here are the most common uses:

torch.nn.functional.relu(*input*, *inplace=False*)
Usage: Applies a relu activation function to an input
Important Parameters: input- the data haha
Example: x = torch.nn.functional.relu(x)

torch.nn.functional.softmax(*input*, *dim=None*, *_stacklevel=3*, *dtype=None*)
Usage: Applies softmax to input data
Important Parameters: dim- dimension to perform softmax. If the data is more than 1d, every slice of data in the specified dimension will sum to 1
Example: torch.nn.functional.softmax(*input*, *dim=2*)

Back to the main stuff now that we know our layers:
There are four main ways to build a neural net in Pytorch: with nn.Module, nn.Sequential, nn.ModuleList, and nn.ModuleDict. Very generally:

- Use Module when you have a big block composed of multiple smaller blocks. This is usually the main class for the neural net.
- Use Sequential when you want to create a small block from layers. This is usually not the main class, but is instead used to define a block of the model that is repeated many times so that it can be easily added to the main class. For simple models, you can also define the whole model using nn.Sequential though.
- Use ModuleList when you need to iterate through some layers or building blocks and do something. This does not define a neural net(it makes no connections), but it stores blocks that can be connected easily in a list.
- Use ModuleDict when you need to parameterize some blocks of your model, for example an activation function. This is used within nn.Module to give individual layers or activation functions flexibility in runtime.

nn.Module approach:

For this approach, we need to create two things within a neural net class: an initialization where we define layers, and a forward function where we describe data flow through the layers.

Init function:

When we build this class. it should inherit from the nn.Module class. Here’s an example init function that doesn’t really do anything.

```
class Net(nn.Module):
   def __init__(self):
      super().__init__() #Here we are calling the init function from the parent class(nn.Module)
      print(“I initialized a net!”) #note that this print statement IS run even when we use the
parent’s init function. This is where we can define our layers!
net = Net()
print(net)
```

```
I initialized a net!
Net()
```

Usually, when you inherit from a parent function, the init function of the parent class(in this case nn.Module) is not run. But, calling super().__init__() instead runs the parent’s init function. If you look at examples online, they might have a syntax like super(Net,self) but that syntax has changed in Python3. As I noted in the comment above, things after the parent init are run. Let’s add layers there:

```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

Forward pass through layers:
Once we have defined our layers, we still need to define how we progress through the layers and any activation functions. This is usually done in a function called forward within the Net() class. An example is shown below. This is a classifier for the CIFAR-10 dataset of 32*32 RGB images with 10 classes:

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5) #parameters are rows, cols. The -1 makes PT figure out how many rows to use. Only one parameter can be -1
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Notice that we were able to do Relu without defining it as a layer(which we could've done with torch.nn in init). As I tried to explain earlier, basic functions are usually implemented with the nn.functional library since they don't have any parameters that need to be initialized. Mostly, this is different activation functions and things like softmax for the head.

In the forward function, x is our input data. So, we are sequentially showing how data flows. Here it goes through our first conv, a relu, our pool, our second conv, a relu, our pool, a reshape to (-1, 400), our first linear layer, a relu, our second linear layer, a relu, and our final linear layer. Below, I tried to show how dimensions change as each image goes through the nn:

Input: 32*32 image with 3 channels (32*32*3)

Layer1: Conv with 6 filters and size 5 kernel
Input: 32*32*3 Output: 28*28*6

Relu: no dimension change

Layer2: maxPool2d
Input: 28*28*6 Output: 14*14*6

Layer3: Conv with 16 filters and size 5 kernel
Input: 14*14*6 Output: 10*10*16

Relu: no dimension change

Layer4: maxPool2d
Input: 10*10*16 Output: 5*5*16

View: Reshape(this is really a flatten layer) to 1*400*1

Layer5: Linear with input 400 and output 120
Relu: no dimension change

Layer6: Linear with input 120 and output 84

Relu: no dimension change

Layer7: Linear with input 84 and output 10

Return: data with dimension 10

Wow, that's cool! The final dimension is 10 and we are looking to classify 10 classes. How convenient. In all seriousness though it's important to make sure you set up a valid data flow. Pytorch will throw an error if you ask it to do something impossible. For example, our conv1 layer could not have stride 2 or an error would be thrown.

nn.Sequential approach:

nn.Sequential adds layers in the order they are declared in its constructor. You can just use this to define the whole model, but it lacks some flexibility since you don’t get to explicitly define the forward method. This makes it hard to do things like skip connections. Here are two examples of how to declare a nn.Sequential model:

```
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

The most useful thing to do with nn.Sequential IMO is to use it for repeated or small blocks. You can add nn.Sequentials to an nn.Module class and treat them like layers:

```
class MyCNNClassifier(nn.Module):
    def __init__(self, in_c, n_classes):
        super().__init__()
        self.conv_block1 = nn.Sequential(
            nn.Conv2d(in_c, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU()
        )

        self.conv_block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )

        self.decoder = nn.Sequential(
            nn.Linear(32 * 28 * 28, 1024),
            nn.Sigmoid(),
            nn.Linear(1024, n_classes)
        )


    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = x.view(x.size(0), -1) # flat
        x = self.decoder(x)
        return x
```

Here, we defined three different blocks within the overall module and still got to create the forward function! If a certain block is repeated many times in the code, you can also create a function for generating these blocks and just call that function repeatedly when defining the layers. One final cool aspect on nn.Sequentials is the ability to generate them from lists. You can create simply python lists of nn layers, and unpack the list into a sequential model. This makes it easy to append new layers and grow nn.Sequentials:

```
modules = []
modules.append(nn.Linear(10, 10))
modules.append(nn.Linear(10, 10))

sequential = nn.Sequential(*modules)
```

An example of both creating a function for generating nn.Sequential blocks and building nn.Sequentials from lists is described in the Resnet example in Section 6.

nn.ModuleList:

nn.ModuleList is a python list that can properly store pytorch modules. This is helpful if you have to iterate through members in the overall model. This example shows how you can iterate through a moduleList, it was not necessary at all to use a moduleList for this network:

```
class NeuralNetwork(nn.Module):
    def __init__(self, n_inputs, n_hidden_unit, n_output):
        super().__init__()
        l1 = nn.Linear(n_inputs, n_hidden_unit)
        a1 = nn.Sigmoid()
        l2 = nn.Linear(n_hidden_unit, n_output)
        s = nn.Softmax(dim=1)

#Make a moduleList from a list of the layers
        l = [l1, a1, l2, s]
        self.module_list = nn.ModuleList(l)

#Iterate through your module to define the forward function
```

```
    def forward(self, x):
        for f in self.module_list:
            x = f(x)
        return x
```

The reason to use nn.ModuleList instead of a normal Python list is that the Modules it contains are visible to all Module methods. If you were to attempt the above with a python list, you'll get an error when you define your optimizer saying that your model has no parameters because Pytorch does not see the parameters of the layers stored in a Python list.

nn.ModuleList has methods similar to python lists! Using these you can do flexible things with your layers. For example, you can create a module that has a variable number of layers:

```
class LinearNet(nn.Module):
#Takes in num_layers as a parameter
  def __init__(self, input_size, num_layers, layers_size, output_size):
     super(LinearNet, self).__init__()

#Define a moduleList
     self.linears = nn.ModuleList([nn.Linear(input_size, layers_size)])

#Use extend() to add all but the last layer to a model. Extend adds all elements of an iterable to the end of a list
     self.linears.extend([nn.Linear(layers_size, layers_size) for i in range(1, self.num_layers-1)])

#Add the last layer with append so that it has the correct output size
     self.linears.append(nn.Linear(layers_size, output_size)
```

There is also an insert method. Here is the documentation(https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html).

nn.ModuleDict:
nn.ModuleDict is a python dictionary that can properly store pytorch modules. Just like the nn.ModuleLists, this means that the modules within it are registered and visible by all Module methods(like the optimizer and such).

moduleDicts are helpful if you need to be able to run your model with different layers or activation functions depending on flags that you pass it. This is mostly used for situations where you want to be able to choose how a layer or activation function looks at runtime. Here is an example:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.choices = nn.ModuleDict({
                'conv': nn.Conv2d(10, 10, 3),
                'pool': nn.MaxPool2d(3)
        })
        self.activations = nn.ModuleDict([
                ['lrelu', nn.LeakyReLU()],
                ['prelu', nn.PReLU()]
        ])

    def forward(self, x, choice, act):
        x = self.choices[choice](x)
        x = self.activations[act](x)
        return x
```

Here, we have a simple neural net class MyModule that inherits nn.Module. Within it we have two ModuleDicts. The first one, self.choices, describes a layer. Depending on how we call forward() when we make our forward pass, the layer is either a 2D conv(if choice parameter is ‘conv’) or a MaxPool(if choice is ‘pool’). The second module dict does a similar thing for activation functions, where you can choose between a leakyRelu and parametricRelu based on the act parameter in the forward() call.

Here is the full documentation(https://pytorch.org/docs/stable/generated/torch.nn.ModuleDict.html).

4. Autograd, backward passes, and loss functions

Autograd is the differentiation engine in torch. In other words, it is the engine that takes care of calculating gradients in backpropogation. This is handy because while we do know how to define neural nets, we still need to be able to train them. Autograd is the first step to that process!

When defining the net, we defined the forward propagation process. In this process, data is run through the series of functions that make up the neural net and are defined by parameters(weights and biases). Pytorch stores these weights and biases in tensors during forward prop.

In backprop, the parameters are adjusted proportionally to the error in the NN guess and the learning rate. This is done by traversing from the output to the input, and calculating local gradients with the chain rule, and optimizing through gradient descent. If the math is confusing, this is pretty good for linear layers and the general idea: https://www.youtube.com/watch?v=tIeHLnjs5U8. You can also find resources for how backprop works on a convolutional layer, but it's hella confusing. I like to think of it generally and the math is just a nice black box. Calling autograd is super easy to do in code:

```
optimizer.zero_grad() #zeros out any remaining gradients, more on optimizers in section 5
loss = my_loss(output, target) #where my_loss is our loss function, more on this below
loss.backward() #call autograd on our loss! This calculates all our local gradients. IMPORTANT> these gradients are immediately available to our optimizer!
```

In the previous paragraph I mentioned updates being proportional to our error. This error comes from a loss function of our choice. Pytorch's torch.nn comes with a ton of prebuilt loss functions. I described the first few, but the rest are in Pytorch docs(https://pytorch.org/docs/stable/nn.html#loss-functions). For all these where I put a definition, x is the true and y is the predicted value:

- Mean Absolute Error Loss- torch.nn.L1Loss
  - This is the average sum of absolute differences between actual and predicted values: loss(x,y) = |x-y|. It is pretty robust to outliers with no squared term and often used for regression.
- Mean Squared Error Loss
  - This is the squared error between predicted and actual values: loss(x,y) = (x-y)^2. It is the default for regression.
- Negative Log-Likelihood Loss
  - This is best for problems with softmax on the output layer. This loss function is L(x,y)=−log(y) for all correct classes(so its minimized when y(our predicted probability)=1).
- Cross-Entropy Loss
  - This computes the difference between two probability distributions for a set of random variables. It specifically punishes strongly for being very confident and incorrect. L(x,y)= -sum(xlog(y))
- Connectionist temporal classification loss
- Negative log likelihood loss with poisson distribution of the target
- Kullback-Leiber divergence loss
- Binary Cross-Entropy loss
- And like 15 more….

We can also define a custom loss function if it doesn't exist in pytorch. It's easy to define a loss function ourselves by just creating a new function! Here's an example:

```
def my_loss(output, target):
   loss = torch.mean((output - target)**2)
   return loss
```

This approach is usually smooth, but it can be tricky in rare situations. If your loss function is not differentiable by autograd, you need to implement the derivative yourself. Autograd can automatically compute derivatives for most variable-based things. If it doesn't work, you can extend autograd by defining forward and backward functions explicitly. This is an example for the LegendrePolynomial function(P(x) = 0.5(5x^3-3x):

```
class LegendrePolynomial3(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

Usually you don’t have to extend autograd though. You can also write loss functions by extending nn.Module. Here are a bunch of examples on Kaggle(https://www.kaggle.com/bigironsphere/loss-function-library-keras-pytorch#Dice-Loss).

Most of the time though, this whole process is pretty simple. We just pick a loss function and call backward() so that autograd calculates gradients.

5. Training, optimization, and checkpoints

The most straightforward part of this section is saving and loading models for checkpoints, so I’ll get that out of the way first. The suggested way is using state dictionaries; they require very little space since they just save parameters. Here’s how:

```
# Saving a Model
torch.save(model.state_dict(), MODEL_PATH)
```

```
# Loading the model checkpoint and loading parameters
checkpoint = torch.load(MODEL_PATH)
model.load_state_dict(checkpoint)
```

If you want to export the model to another environment(like tensorflow) for inference, there will be a different process for converting to the required format.

The next step is picking an optimizer. The optimizer steps in the direction of the gradients based on some algorithm. Every optimizer in pt shares a base class torch.optim.Optimizer, but the different types of optimizers have unique init() and step() classes. Pytorch supports the following natively. All the links go to a really cool website that gives a description of each optimizer:

- AdaDelta Class
- AdaGrad Class
- Adam Class
- AdamW Class
- SparseAdam Class
- Adamax Class
- LBFGS class
- RMSprop class
- Rprop class
- SGD Class
- ASGD class

There is also a library torch_optimizer that supports:

- AccSGD
- AdaBound
- AdaMod
- Adafactor
- AdamP
- AggMo
- DiffGrad
- Lamb
- NovoGrad
- PID
- QHAdam
- QHM
- RAdam
- SGDP
- SGDW
- Shampoo
- SWATS
- Yogi

Deciding which optimizer to use is a matter of a lot of experience and a lot of work to understand what each one does. I'm definitely not there yet. Generally, all of the optimizers are derived from Gradient Descent, with some modification to improve convergence. Also, the most common optimizer seems to be Adam or AdamW.

Unnecessary but cool Adam/AdamW description:

Adam extends gradient descent by using past gradients. The general strategy is to take larger steps when gradients change slowly and smaller steps when they change rapidly. These step sizes are determined for each weight individually.

To get farther into the math, Adam makes observations about the gradient using the exponential moving average of the gradient(this is called the first moment m(t)) and the square of the gradients(called the second moment v(t)). The exponential moving average is very similar to a simple moving average, but it places more weight on recent observations:

$$EMA_t = (x * \frac{s}{1+w}) + EMA_{t-1} * (1 - (\frac{s}{1+w}))$$

Where EMA_t is the current exponential moving average, x is the current measurement, s is a smoothing factor, w is the amount of samples in the window, and EMA_{t-1} is the previous exponential moving average

In the equations for Adam, they use two variables β1 and β2 to describe the smoothing and window size for the first moment m(t) and second moment v(t) respectively:

By default, β1 = 0.9 = $(1 - (\frac{s}{1+w}))$ for the first moment m(t)

And β2 = 0.999 = $(1 - (\frac{s}{1+w}))$ for the second moment v(t)

Plugging this into the EMA equation, we get the equations used in Adam as the following. m(t) and v(t) are initialized to 0 for the first time step:

$m(t) = \beta1 \cdot m(t-1) + (1-\beta1) \cdot g(t)$

$v(t) = \beta2 \cdot v(t-1) + (1-\beta2) \cdot g(t)^2$

The issue with this conventional approach to an EMA is that you can’t calculate a moving average until many observations are made. This is easiest to see in the original EMA equation. In that equation, until w+1 observations are made, you can’t calculate EMA. Even though we did a change of variable to β1 and β2, this issue persists. You can prove this looking at the first time step:

m(1) = 0.9 · 0 + 0.1 · *g(1)* = 0.1 · *g(1)* *#keep in mind that m(0) is initialized to 0!*

So, our equation says that the average of our one gradient is 1/10 of that gradient :/. While the conventional EMA solution to this problem is to wait for w+1 observations, this doesn't work in a neural net where we need a large window and to update immediately. The solution used by Adam is to rescale m(t) and v(t) by (1-β^t)*,* where t is the timestep:

m(t) = (β1 · m(t-1) + (1-β1) · g(t))/(1-β1^t)

v(t) = (β2 · v(t-1) + (1-β2) · $g(t)^2$)/(1-β2^t)

To update a weight, Adam uses the following equation:

x(t)= x(t-1) — α · m(t) / [sqrt(v(t)) + ϵ]

Where x(t) is the new weight, α is the learning rate, and ϵ is 10^{-8} to avoid dividing by 0.

Intuition behind step:

The variance of a random variable *x* is defined as *Var(x)* = $<x^2>-<x>^2$ where < > is the expected value. Here's a little proof, where E[thing] is expected value of thing:

$$\begin{aligned}\operatorname{Var}(X) &= \operatorname{E}\left[(X-\operatorname{E}[X])^2\right] \\ &= \operatorname{E}\left[X^2 - 2X\operatorname{E}[X] + \operatorname{E}[X]^2\right] \\ &= \operatorname{E}\left[X^2\right] - 2\operatorname{E}[X]\operatorname{E}[X] + \operatorname{E}[X]^2 \\ &= \operatorname{E}\left[X^2\right] - \operatorname{E}[X]^2\end{aligned}$$

The exponential moving average of the square of the gradients *v(t)* is called the uncentered variance because we did not subtract the square of the mean of the gradients.

The variance quantifies how much the gradients vary around their means. If the gradients stay approximately constant and we want to take a large step, the variance of the gradients is approximately 0 and the uncentered variance *v(t)* approximately equal to $m(t)^2$. This means that *m(t) / sqrt(v(t))* is around 1 and the step is in the order of *α*.

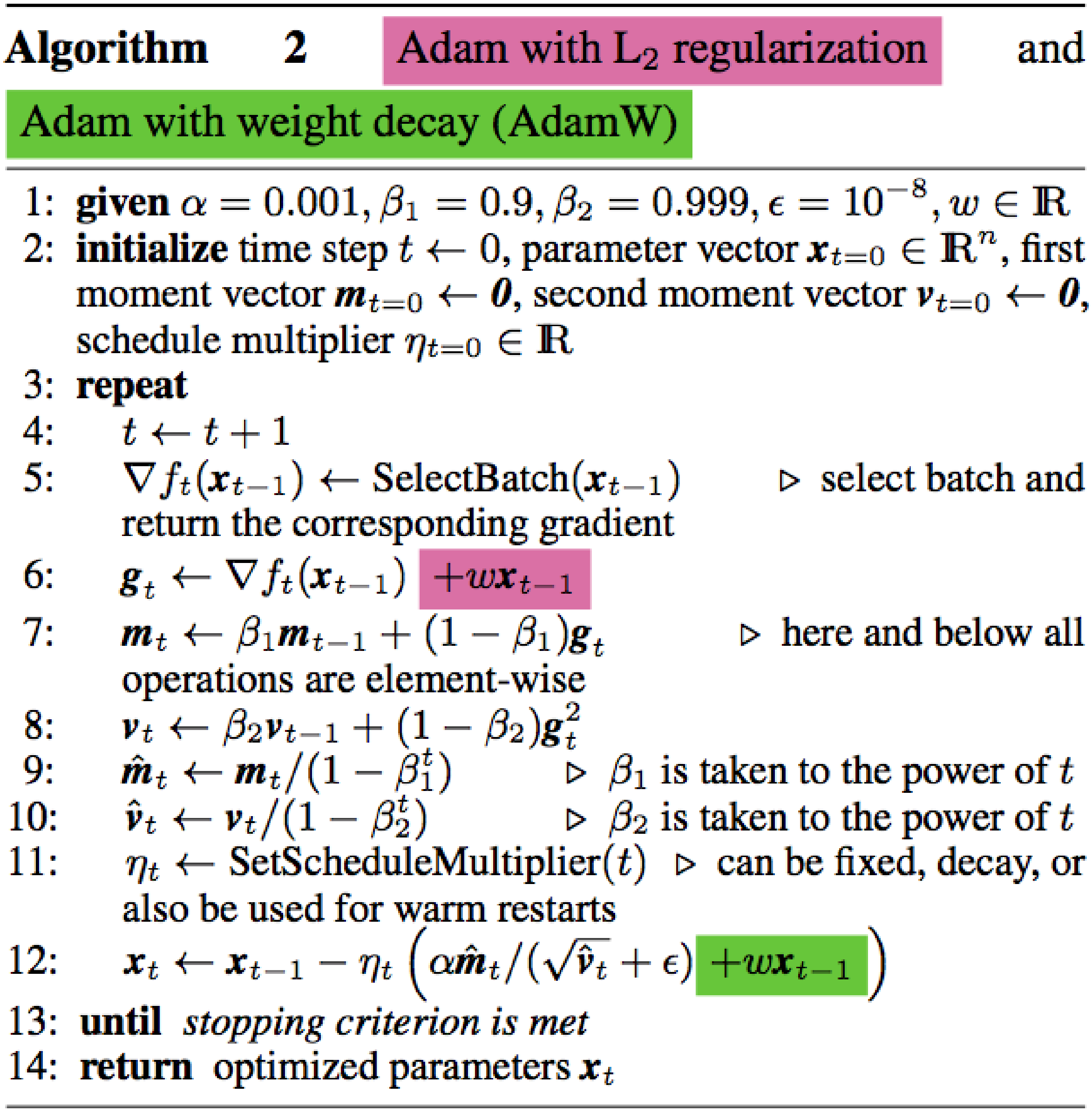
If on the other hand, the gradients are changing rapidly, *sqrt(v(t))* is much larger than *m(t)* and the step is therefore much smaller than *α*.

Regularization:

Adam also often implements L2 regularization or weight decay to promote finding small weights. The reasoning for this is that networks with smaller weights are observed to overfit less and generalize better. This idea is crazy, but it is explained well with cool visuals here:

http://neuralnetworksanddeeplearning.com/chap3.html#why_does_regularization_help_reduce_overfitting

When we add regularization to Adam, we get the following algorithm:



Algorithm 2 Adam with L$_2$ regularization and Adam with weight decay (AdamW)

```
given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
    $t \leftarrow t + 1$
    $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$    ▷ select batch and return the corresponding gradient
    $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1})$ $+w\boldsymbol{x}_{t-1}$
    $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$    ▷ here and below all operations are element-wise
    $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
    $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1 - \beta_1^t)$    ▷ $\beta_1$ is taken to the power of $t$
    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1 - \beta_2^t)$    ▷ $\beta_2$ is taken to the power of $t$
    $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$    ▷ can be fixed, decay, or also be used for warm restarts
    $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) +w\boldsymbol{x}_{t-1}\right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{x}_t$
```

The violet term represents adding L2 regularization. Now, the moving average keeps track of both the gradients and regularization.

Adding the green term creates a new optimizer called AdamW. This approach adds another regularization term that is not divided by the term [sqrt(v(t)) + ϵ] in the new weight calculation.

This makes it so that weights with fast-changing gradients(and therefore very large values of [sqrt(v(t)) + ϵ]) are also properly regularized.

Back to the main stuff:

To use one of the Pytorch optimizers, it is pretty straightforward:

Outside of the training loop:

```
optimizer = Adam(model.parameters(), lr=0.07) # Define the optimizer of choice
```

Inside the training loop:

```
optimizer.zero_grad() #delete any stored gradients

loss_train = criterion(output_train, y_train) #Compute a loss from your loss function

loss_train.backward() #compute gradients based on loss

optimizer.step() #use the step method to update weights based on the new gradients
```

You can also make your own optimizer. It has to inherit from the base optimizer class and implement a custom init() and step() method. I have never seen it done, but here is a good link if it comes up: (http://mcneela.github.io/machine_learning/2019/09/03/Writing-Your-Own-Optimizers-In-Pytorch.html).

One final detail before training. Models have a method model.train(). This method tells the model that training is about to begin and sets it in a training mode. This is important for layers like dropout that have different training and test behavior. So, it should be called before the training loop.

Putting together a bunch of sections, we can build the training loop. There are a few main components:

1. Call model.train()
2. Step through epochs
3. Load data-Section2
4. Forward pass-Section3
5. Loss-Section4
6. Calculate gradients-Section4
7. Step with optimizer-Section5

```
#set model in training mode and iterate through the epochs
model.train()
for epoch in range(num_epochs):
#load data if needed. This is from Section2. In this example we are using a dataloader
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
 #Forward pass through model. Section 3
        outputs = model(images)
#Calculte the loss with the loss function. Section 4
        loss = criterion(outputs, labels)

        #Zero out any gradients stored in the optimizer. Section 5
        optimizer.zero_grad()
        #Backward pass. This loads new gradients. Section 4
        loss.backward()
        #Step with new gradients to update weights. Section 5
        optimizer.step()
```

This can also be implemented as a function and called every epoch. Here’s a different example of that:

```
def train(epoch):
    model.train()
    tr_loss = 0
    # getting the training set. Here there is not a dataloader. The training and validation sets are
just tensors and instead of batching we are just feeding the whole set to train every time. This is
not super common
    x_train, y_train = Variable(train_x), Variable(train_y)
    # getting the validation set
    x_val, y_val = Variable(val_x), Variable(val_y)
    # converting the data into GPU format
    if torch.cuda.is_available():
        x_train = x_train.cuda()
        y_train = y_train.cuda()
        x_val = x_val.cuda()
        y_val = y_val.cuda()

    # clearing the Gradients of the model parameters
    optimizer.zero_grad()

    # prediction for training and validation set
    output_train = model(x_train)
    output_val = model(x_val)
    # computing the training and validation loss
    loss_train = criterion(output_train, y_train)
    loss_val = criterion(output_val, y_val)
#here we decide to save losses for graphing
    train_losses.append(loss_train)
    val_losses.append(loss_val)

    # computing the updated weights of all the model parameters
    loss_train.backward()
    optimizer.step()
    tr_loss = loss_train.item()
    if epoch%2 == 0:
        # printing the validation loss
        print('Epoch : ',epoch+1, '\t', 'loss :', loss_val)
```

6. Evaluating a model

This section is short yay. When we want to evaluate the model, we first have to set it in evaluation mode. This is for layers like dropout that have different behavior during evaluation and training:

```
model.eval() #set the model in evaluation mode
```

Then, we just need to feed the model data! This can look different depending on how your test images are stored. Here are two examples:

Example 1: No dataloader and classifying images from multiple classes

```
# loading test images. The ids are stored in a csv in this case
test_img = []
#iterate through the csv id category
for img_name in tqdm(test['id']):
    # defining the image path
    image_path = 'test_ScVgIM0/test/' + str(img_name) + '.png'
    # reading the image
    img = imread(image_path, as_gray=True)
    # normalizing the pixel values
    img /= 255.0
    # converting the type of pixel to float 32
    img = img.astype('float32')
    # appending the image into the list of test images
    test_img.append(img)

# converting the list to numpy array
```

```
test_x = np.array(test_img)
test_x.shape

# converting training images into torch tensor
test_x = test_x.reshape(10000, 1, 28, 28)
test_x  = torch.from_numpy(test_x)
test_x.shape

# generating predictions for test set
with torch.no_grad():
    output = model(test_x.cuda())
#This example model outputs a 1d tensor of activations for each class. We use softmax to get probabilities and the max probability is the class estimate
softmax = torch.exp(output).cpu()
prob = list(softmax.numpy())
predictions = np.argmax(prob, axis=1)
#predictions is not a list of predicted class id's!
```

Example 2 with dataloader:

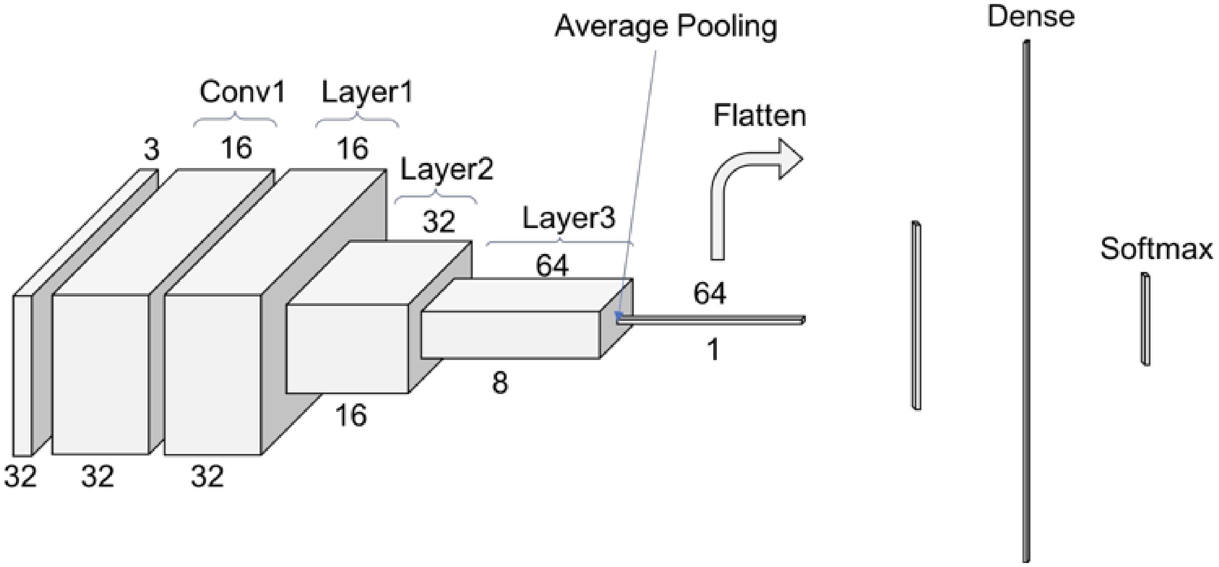
```
model.eval()
#Torch.no_grad() tells Pt that it doesn't need to calculate gradients. This is useful for inference since it reduces memory consumption
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the test images: {} %'.format(100 * correct / total))
```

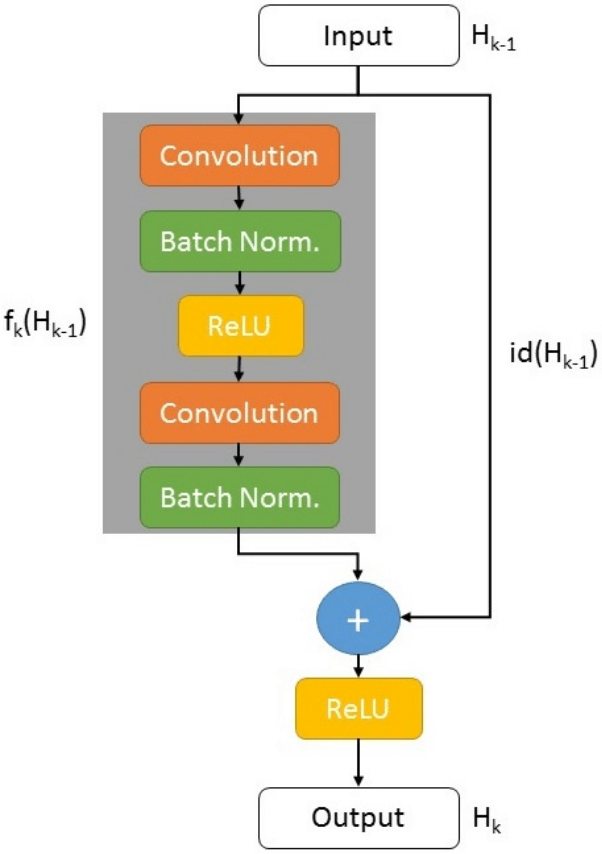
7. Example full code structure

Here is an example for a U-net implementation. The programmer split up the different parts(training, loss function, loss evaluation, model) but they should all be pretty familiar if you read this whole garbage: https://github.com/milesial/Pytorch-UNet. Still, it's pretty complex.

Here's another example that's all in one place for the deep residual network on the CIFAR-10 dataset. I took this example from this github with many different level pt implementations: https://github.com/yunjey/pytorch-tutorial. Check it out for more examples, but it doesn't have any explanations of model structure or the most detailed comments. Anywho, the structure is shown below:



Where each Layer is built from two residual blocks. These residual blocks have two convolutions each along with a path for data to skip the convolutions. Layer 1 has a stride of 1 for all four convolutions, maintaining the input size. Layers 2 and 3 have a stride of 2 for the first convolution and also add a convolution with stride 2 on the skip connection for downsampling the data. A residual block without the downsampling on the skip connection is shown below:



This link describes a very similar Resnet(https://towardsdatascience.com/resnets-for-cifar-10-e63e900524e0). The only difference is that the example implementation here has a skip connection for each block(2 per layer), while the link only describes one skip connection per layer. The rest of the model structure is the same.

Since this is not the simplest CNN, I added many comments to this code I ripped to make it easier to understand:

```
# --------------------------------------------------------------------------- #
# An implementation of https://arxiv.org/pdf/1512.03385.pdf                    #
# See section 4.2 for the model architecture on CIFAR-10                       #
# Some part of the code was referenced from below                              #
# https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py   #
# --------------------------------------------------------------------------- #

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device configuration. This is from the first section, if it is available, we want to use our GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters. These are our training parameters. Briefly described in section 5
num_epochs = 80
batch_size = 100
learning_rate = 0.001

# Image preprocessing modules. This sort of thing is described in section 2. The random flops
and crops augment our dataset to make it more robust to overfitting
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])

# CIFAR-10 dataset. Load the dataset provided by torchvision. This is in section 2
train_dataset = torchvision.datasets.CIFAR10(root='../../data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

test_dataset = torchvision.datasets.CIFAR10(root='../../data/',
                                            train=False,
                                            transform=transforms.ToTensor())

# Data loader. Create a data loader for feeding batches from CPU to GPU. This is also in
Section 2
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)

# 3x3 convolution. Create a new block conv3x3 that we can use in our model. I didn’t explicitly
talk about this in here, but it builds off of the layers and model construction in Section 3. This is
done just for convenience as we will be using a lot of convs of this type
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)

# Residual block. This is the same idea as the conv3x3 block above. However, here, we are
describing a sub-model. This is a structure in our overall model that shows up very often(6
times) so its easier to make it as a model and initialize it into the model as needed.

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# ResNet. Here we describe the full model with the initial conv, 3 layers, and pooling with linear
layer at the end. This is similar to section 3
class ResNet(nn.Module):
    def __init__(self, blokck, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)
```

```
 # Make layers. This section defines a function for making the Resnet block layers. It’s the hardest to understand, so I commented thoroughly
    def make_layer(self, block, out_channels, blocks, stride=1):
#By default, we don’t downsample. For the layers that downsample, we first declare the downsample layer for the skip connection.
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
#We then create a list for our residual blocks that will build the layer. The first block in the layer has the downsample parameter. Keep in mind that in_channels won’t equal out_channels if it is downsampling, so the dimensions match between the normal conv path and skip connection of the residual block
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
#Set in_channels to out_channels so no more layers downsample and add the rest of the residual blocks. For our standard approach of 2 blocks per layer, this just adds one more block
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
#Convert our list of residual blocks to a sequential model so that it can be easily incorporated into our full Resnet model(can’t just add lists to an nn.Module).
        return nn.Sequential(*layers)

#Describe forward path through resnet
    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

#initialize model with 2 blocks per layer(the list [2,2,2] describes blocks for each layer). Initialize it on cuda also if possible
model = ResNet(ResidualBlock, [2, 2, 2]).to(device)
```

```
# Loss and optimizer. In this case, loss and optimizer are both built into pytorch(Sections 4 and
5)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# For updating learning rate. Create a function for updating the learning rate. The paper asks us
to lower the learning rate as we go, so this makes it easier. There are optimizers that adjust the
learning rate, but this programmer wanted to do it manually to match the paper
def update_lr(optimizer, lr):
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

# Train the model. This is similar to section 5. Iterate through epochs and within each epoch get
images from the data loader and pass them to the GPU if CUDA, CPU otherwise.
total_step = len(train_loader)
curr_lr = learning_rate
model.train()
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass of training images through the model. Calculate the loss with our loss
function name criterion. Section 5
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize. Zero out gradients, calculate them with autograd, and then use
the optimizer to update weights. Section 5/4
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

#Print an update every 100 batches
        if (i+1) % 100 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

    # Decay learning rate
    if (epoch+1) % 20 == 0:
        curr_lr /= 3
        update_lr(optimizer, curr_lr)
```

```
# Test the model. Test accuracy by running inference on the validation set on the GPU. Section 5
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the test images: {} %'.format(100 * correct / total))

# Save the model checkpoint. Section 5
torch.save(model.state_dict(), 'resnet.ckpt')
```

8. Pretrained models

Another thing to note is that pytorch comes with a gigantic library of pretrained models. These can be loaded and retrained on custom data, or used as a starting point for small layer manipulations. Here is the torchvision.models library(https://pytorch.org/docs/stable/torchvision/models.html).

To use these models, we do our normal imports. We can then import a model using the Pytorch constructor, and can specify whether we want the pretrained version.

```
import torch.nn as nn
import torch
import torch.autograd
import torchvision.models as models

Model = models.resnet18()
pretrainedModel = models.alexnet(pretrained=True)
```

Pretrained models are very handy even if your problem is seemingly unrelated. This is the case because it will converge more quickly on a smaller dataset. This process is called transfer learning(https://cs231n.github.io/transfer-learning/). But, how do we manipulate these models? I'll go over a few common things:

1. Freezing a model or certain layers

Freezing a piece or whole model locks in all trainable parameters so that they cannot be changed. To access parts of the model for freezing, we can use the .children() function:

```
for child in model.children()
print(child)
```

Children in the pretrained model can be individual layers:

```
Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
```

Or blocks:

```
Sequential (
  (0): BasicBlock (
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
    (relu): ReLU (inplace)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  )
  (1): BasicBlock (
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
    (relu): ReLU (inplace)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True)
  )
)
```

We can access the parameters of a child using:

```
for param in child.parameters():
        print("This is what a parameter looks like - \n",param)
```

```
This is what a parameter looks like -  Parameter containing:
(0 ,0 ,.,.) =
  1.8160e-02  2.1680e-02  5.6358e-02  ...  -1.2987e-02 -6.1262e-02 -4.8870e-02
  2.6440e-02  1.0603e-02  1.9794e-02  ...  -4.2643e-02 -4.5565e-03 -4.8300e-02
  9.0205e-03  1.9536e-03  1.9925e-04  ...   1.1413e-02  1.1395e-02  2.8418e-03
                ...                ⋱                ...
 -2.4830e-02  8.1022e-03 -4.9934e-02  ...   2.2573e-02  1.6346e-02  3.9666e-02
 -2.3857e-02 -1.6275e-02  2.9058e-02  ...   3.0488e-02  2.0294e-02 -5.1073e-03
 -1.6848e-04  5.9266e-02 -5.8456e-03  ...   1.9757e-02 -7.8441e-02  1.3667e-02
…
```

Freezing certain layers can make training much faster(lots less calculations). We can freeze parameters by setting their param.reqires_grad to false. This code block freezes the parameters up to the second layer of child 6. Children can have layers if they are blocks:

```
child_counter = 0
for child in model.children():
    if child_counter < 6:
        print("child ",child_counter," was frozen")
        for param in child.parameters():
            param.requires_grad = False
    elif child_counter == 6:
        children_of_child_counter = 0
        for children_of_child in child.children():
            if children_of_child_counter < 1:
                for param in children_of_child.parameters():
                    param.requires_grad = False
                print('child ', children_of_child_counter, 'of child',child_counter,' was frozen')
            else:
                print('child ', children_of_child_counter, 'of child',child_counter,' was not frozen')
            children_of_child_counter += 1

    else:
        print("child ",child_counter," was not frozen")
    child_counter += 1
```

```
child  0  was frozen
child  1  was frozen
child  2  was frozen
child  3  was frozen
child  4  was frozen
child  5  was frozen
child  0 of child 6  was frozen
child  1 of child 6  was not frozen
child  7  was not frozen
child  8  was not frozen
child  9  was not frozen
```

We have to also make sure that we call the optimizer on just the trainable parameters(now that some are frozen). We can do this by filtering for requires grad:

Bad if parts are frozen:

```
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.1)
```

Good if parts are frozen:

```
optimizer = torch.optim.RMSprop(filter(lambda p: p.requires_grad, model.parameters()), lr=0.1)
```

2. Changing layers

One way to change a layer is by overwriting it. This is common in object detection problems where you just need to change the size of the last layer for a new number of output classes.

```
# Get number of parameters going in to the last layer. we need this to change the final layer. In our case, this layer is called model.fc
num_final_in = model.fc.in_features

#overwrite for 300 classes
NUM_CLASSES = 300
model.fc = nn.Linear(num_final_in, NUM_CLASSES)
```

We can also convert the children to a list and slice out layers as needed. Then, we can convert the list back into a sequential model. You can also add in layers to the list before conversion.

```
#These examples remove the final and final two layers
new_model = nn.Sequential(*list(model.children())[:-1])
new_model_2_removed = nn.Sequential(*list(model.children())[:-2])
```

The most common way to add layers though is by modifying the pytorch models via adding layers. Here are two examples:

Ex 1. Modify the Resnet model for 300 classes, load parameters saved on the pc, freeze the first half, and add three layers. To do his, we are creating a new class and importing the model within it.

```
import torch.nn as nn
import math
import torch.utils.model_zoo as model_zoo
import torch
from torch.autograd.variable import Variable
from torchvision import datasets, models, transforms


class Resnet_Added_Layers_Half_Frozen(nn.Module):
    def __init__(self,LOAD_VIS_URL=None):
        super(ResnetCombinedFull2, self).__init__()

         # Load the resnet model. Change the amount of output classes and load parameters from a checkpoint
        model = models.resnet18(pretrained = False)
        num_final_in = model.fc.in_features
        model.fc = nn.Linear(num_final_in, 300)
        checkpoint = torch.load(MODEL_PATH)
        model.load_state_dict(checkpoint)

#Freeze through the second child of the sixth layer. This point is arbitrary, just used as an example
```

```
        for child in model.children():
            if child_counter < 6:
                for param in child.parameters():
                    param.requires_grad = False
            elif child_counter == 6:
                children_of_child_counter = 0
                for children_of_child in child.children():
                    if children_of_child_counter < 1:
                        for param in children_of_child.parameters():
                            param.requires_grad = False
                    else:
                    children_of_child_counter += 1

            else:
                print("child ",child_counter," was not frozen")
            child_counter += 1

        # Define the pretrained model and any layers we want to add
        self.vismodel = nn.Sequential(*list(model.children()))
        self.projective = nn.Linear(512,400)
        self.nonlinearity = nn.ReLU(inplace=True)
        self.projective2 = nn.Linear(400,300)

    #describe the data flow through the model and our extra layers
    def forward(self,x):
        x = self.vismodel(x)
        x = torch.squeeze(x)
        x = self.projective(x)
        x = self.nonlinearity(x)
        x = self.projective2(x)
        return x
```

Ex 2. Create a resnet with two new layers by deriving from Resnet and adding our layers. Here, our class directly derives from resnet, and we just add a couple layers. To see how this compares to the source resnet implementation, you can look at the code here: (https://pytorch.org/docs/stable/_modules/torchvision/models/resnet.html#resnet34). The model is initialized the same way, but our new model has a slightly different forward function using the new layers:

```
class MyResnet2(models.ResNet):
    def __init__(self, block, layers, num_classes=1000):
        super(MyResnet2, self).__init__(block, layers, num_classes)
        self.conv_feat = nn.Conv2d(in_channels=512,
                                   out_channels=6000,
                                   kernel_size=1)
        self.fc = nn.Linear(in_features=6000,
                            out_features=6000)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = self.conv_feat(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x

from torchvision.models.resnet import BasicBlock
model = MyResnet2(BasicBlock, [3, 4, 6, 3], 1000)

#Example of running it on random data
x = Variable(torch.randn(1, 3, 224, 224))
output = model(x)
```

9. Whats a references?

I ripped code and stuff from so many places and didn’t keep track :O