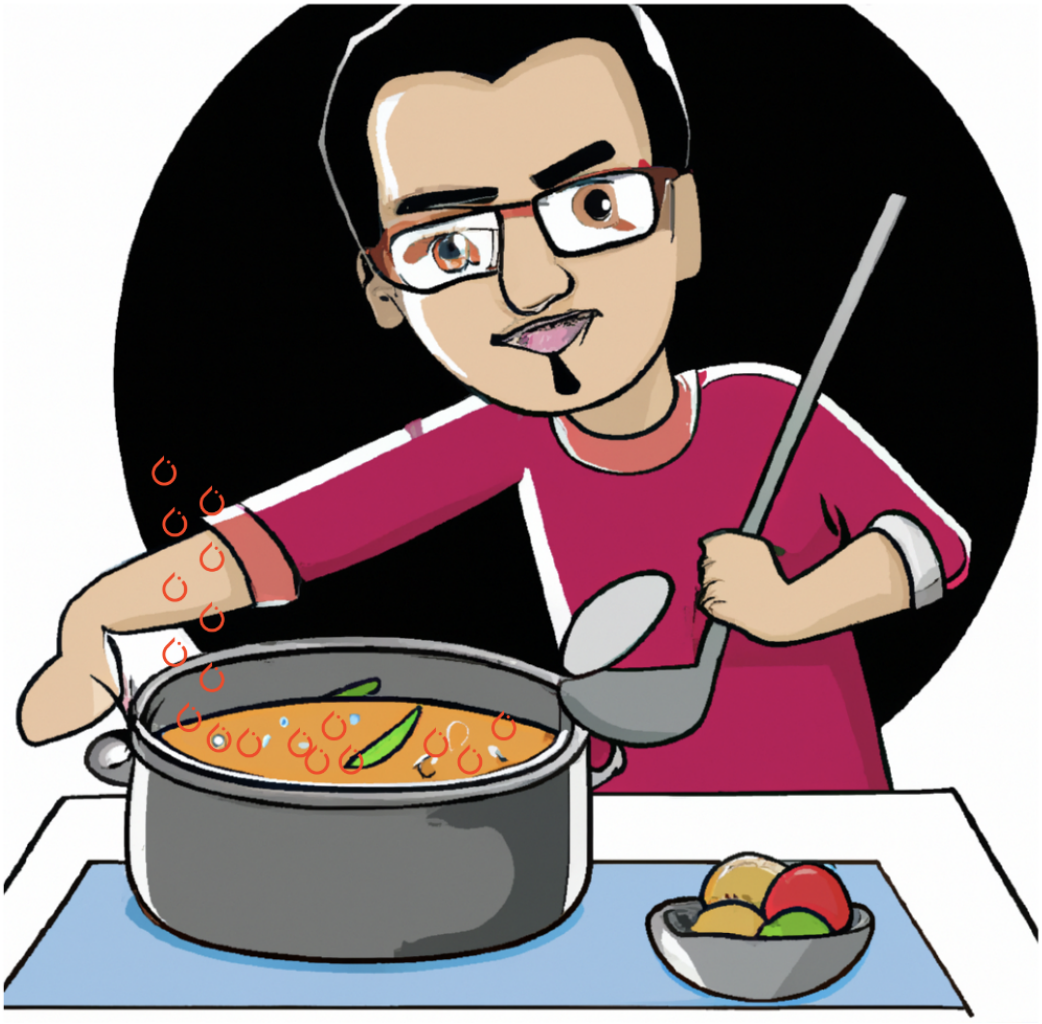
The PyTorch 101 Cookbook



The 3-Steps "no hassle" Recipe to Build and Train Models with PyTorch, even if you're not used to it!

The Road to PyTorch?

You probably invested a lot of time in PyTorch already. But if you've been unsuccessful so far, let me try to give you the main elements to have a network up and running. These will be reexplained as we code them, but it's good if you see them right now so you don't feel shocked later in the workshop:

When you're not coming from PyTorch (Keras?), the main things to know are:

- Datasets & DataLoaders
- Init & Forward
- Training & Running

Datasets & DataLoaders

Let’s begin with the Dataset.

Dataset

This is the minimal code of any PyTorch Dataset class:

```
class BDD100k_dataset(class):
  def __init__(self):
    pass
  def __len__(self):
    pass
  def __getitem__(self):
    pass
```

In our case, we'll fill our class this way:

```
class BDD100k_dataset(Dataset):
    def __init__(self, images, labels, tf=None):
        """Dataset class for BDD100k"""
        self.images = images
        self.labels = labels
        self.tf = tf

    def __len__(self):
        return self.images.shape[0]

    def __getitem__(self, index):
        # read source image and convert to RGB, apply transform
        rgb_image = self.images[index]
        if self.tf is not None:
            rgb_image = self.tf(rgb_image)

        # read label image and convert to torch tensor
        label_image  = torch.from_numpy(self.labels[index]).long()
        return rgb_image, label_image
```

Let’s try to understand our 3 main functions:

- Init
- Len
- Getitem

The **Init** Function

```
def __init__(self, images, labels, tf=None):
        self.images = images
        self.labels = labels
        self.tf = tf
```

In init, we define the variables and transformations. In PyTorch, transformations are used to normalize the data, convert an image to a PyTorch tensor (mandatory), or even do data augmentation.

For example:

```
preprocess = transforms.Compose([
transforms.ToTensor(), transforms.Normalize(mean=(0.485, 0.56, 0.406),
                                            std=(0.229, 0.224, 0.225))])
data = BDD100k_dataset(images, labels, tf=preprocess)
```

The **Len** Function

```
    def __len__(self):
        return self.images.shape[0]
```

In len, we simply return the length of our dataset. This can be used later to loop through the whole dataset when training.

We need to define this function because the BDD100K_dataset class isn't something PyTorch knows. The type of the object isn't a list, or a string, it's a BDD100K_dataset! So you need to make sure than when we do len(training_set), we have a result that corresponds to the number of training samples.

An example use would be defining a training and test set:

```
total_count = len(data)
train_count = int(0.7 * total_count)
valid_count = int(0.2 * total_count)
test_count = total_count - train_count - valid_count
train_set, val_set, test_set = torch.utils.data.random_split(data,
            (train_count, valid_count, test_count),
             generator=torch.Generator().manual_seed(1))
```

The **GetItem** Function

```
    def __getitem__(self, index):
        # read source image and convert to RGB, apply transform
        rgb_image = self.images[index]
        if self.tf is not None:
            rgb_image = self.tf(rgb_image)

        # read label image and convert to torch tensor
        label_image  = torch.from_numpy(self.labels[index]).long()
        return rgb_image, label_image
```

In getitem, we return the image and label at a specific index. We won't use __getitem__ directly as a function, but we will want to do training_set[138] and need to get the image and label at this index.

This is a tricky part, because it requires us to return both the image and the label. If you're doing multi-task learning, you'll want to return several labels!

For example, in my HydraNets course, we're learning to predict the age, gender, and race from an image. One network does 3 predictions, and therefore one piece of data has 3 labels!

```
    def __getitem__(self, index):
        # Load an Image
        img = Image.open(self.images[index]).convert('RGB')
        # Transform it
        img = self.transform(img)
        # Get the Labels
        age = self.ages[index]
        gender = self.genders[index]
        race = self.races[index]

        return img, age, gender, race
```

Example use: Visualizing an element of the dataset

```
sample_image, sample_label = train_set[0]
plt.imshow(sample_image)
plt.show()
```

So this was for the Dataset, then there is the DataLoader.

DataLoader

A dataset contains your data, a DataLoader is the PyTorch way to load the Dataset. To do that:

```
train_dataloader = DataLoader(train_set, batch_size=8)
val_dataloader   = DataLoader(val_set, batch_size=8)
test_dataloader  = DataLoader(test_set, batch_size=8)
```

Once you have this, you need to create a model.

The Model

If you're used to Keras, you probably defined models differently from what we'll do now. For example, in the v1 of my segmentation course, we defined an encoder-decoder architecture with Keras this way (skipping the imports):

```
def simple_encoder_decoder(input_shape, pool_size):
    model=Sequential()
    model.add(BatchNormalization(input_shape=input_shape))
    model.add(Conv2D(8, (5,5), padding='valid', strides=(1,1), activation='relu'))
    model.add(Conv2D(8, (5,5), padding='valid', strides=(1,1), activation='relu'))
    model.add(MaxPooling2D(pool_size=pool_size))
    model.add(UpSampling2D(size=pool_size))
    model.add(Conv2DTranspose(8, (5,5), padding='valid', strides=(1,1), activation='relu'))
    model.add(Conv2DTranspose(3, (5,5), padding='valid', strides=(1,1), activation='relu'))
    model.summary()
```

This was simple, and elegant.

In PyTorch, the implementation looks different, here is the minimal code:

```
class ModelToBuild(nn.Module):
    def __init__(self, net):
        pass

    def forward(self, x):
        pass
```

Again, we have 2 functions to understand:

- Init
- Forward

It may look harder, but we'll dive into these 2 functions and you'll see that it's actually a smart way to implement networks.

Let's begin with the init() function.

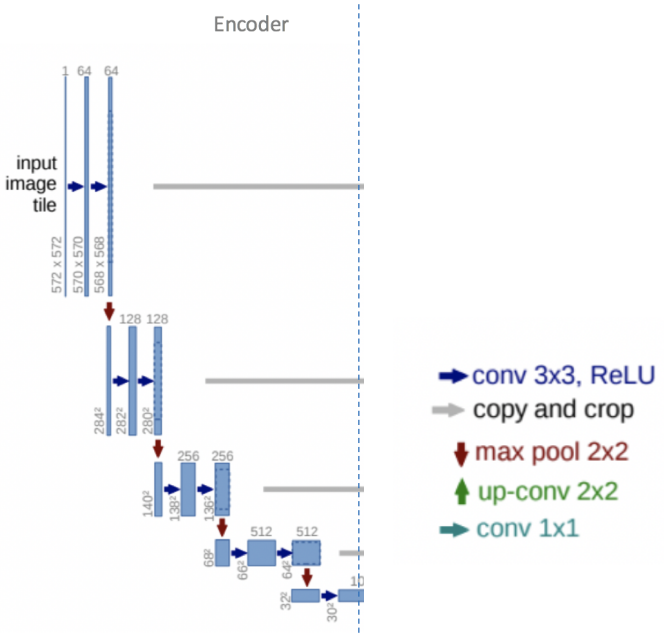
The Init() Function

PyTorch works in an ingredient/salad making way. The init function is where you define the ingredient, the forward() function is the salad, the addition of these ingredients together.

In the example above, notice how we have several operations that repeat themselves: Convolutions, BatchNorm, MaxPooling, Upsampling, Transposed Convolutions, etc…

PyTorch decides to define these operations first, and then to use them. So if you have a series of 30 convolutions, you can simply use a for loop.

For example, let's say we want to implement the Encoder in UNet. The Encoder is just a series of Double Convolutions, followed by a MaxPooling every time.



The UNet encoder is a series of 4 double convolutions

In code, it would look like this:

```
class UNetEncoder(nn.Module):
    def __init__(self, in_channels, out_channels, layer_channels):
        super(UNetEncoder, self).__init__()
        self.encoder = nn.ModuleList()

        # Double Convolution
        for num_channels in layer_channels:
            self.encoder.append(double_conv(in_channels, num_channels))
            in_channels = num_channels

        # Pooling
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
```

The double_conv function is defined separately for easier understanding, it looks like this:

```
def double_conv(in_channels, out_channels):
   return nn.Sequential(nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride =1, padding = 1, bias=False),
       nn.BatchNorm2d(out_channels),
       nn.ReLU(inplace=True),
       nn.Conv2d(out_channels, out_channels, kernel_size=3, stride =1,
padding = 1, bias=False),
       nn.BatchNorm2d(out_channels),
       nn.ReLU(inplace=True),
   )
```

So the **init()** function really defines what we'll do, and the **forward()** function will define how we do it.

The Forward() Function

The forward function simply tells in which order to call the functions.

For example:

```
    def forward(self):
        # Pass input image through Encoder
        for down in self.encoder:
            x = down(x)
            x = self.pool(x)
        return x
```

And that's it!

Now, to call a model, you simply run:

```
model = UNetEncoder(in_channels=3, out_channels=3,
layer_channels=[64, 128, 256, 512]).to(device)
```

The **.to()** function sends the model to either the GPU or the CPU.

Next, let's see the training loop, and run code!

The Training Loop, and Run Code

We'll first look at how to train a model, and then how to use it to run.

The Training Loop

The minimal training code for training looks like this:

```
for epoch in range(n_epochs):
    model.train() # Set model to training

    for i, data in enumerate(train_dataloader):
        input = ### GET INPUT FROM "DATA"
        optimizer.zero_grad() #NO GRADIENTS
        output = model(input) #INFERENCE
        loss_1 = ### GET LOSS
        loss.backward() #COMPUTES THE GRADIENTS DURING BACK PROPAGATION
        optimizer.step() # UPDATE THE WEIGHTS
```

Notice the different steps we need to implement:

1. Loop through the **DataLoader**
2. Collect our **input**
3. Run a **forward** pass
4. Calculate the **loss**
5. **Backpropagate**

In the middle, we also have optimizer operations, such as **.zero_grad()** that freezes the gradients (to run an inference), and **.step()** that increments the learning from 1 item.

An example implementation would look like this. Try to identify the steps we listed above.

```
for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        for inputs, labels in tqdm(dataloader_train,
total=len_train_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass
            y_preds = model(inputs)
            loss = criterion(y_preds, labels)
            train_loss += loss.item()

            # Backward pass
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            # adjust learning rate
            if lr_scheduler is not None:
                lr_scheduler.step()

        # compute per batch losses, metric value
        train_loss = train_loss / len(dataloader_train)
        validation_loss, validation_metric = evaluate_model(
             model, dataloader_valid, criterion, metric_class,  num_classes,
device)
```

And of course, to call it, simply run a function that encapsulates this code.

Run

Finally, how do we run a model?

It couldn't be simpler, you just:

```
    model.eval()
    with torch.no_grad():
            inputs = inputs.to(device)
            y_preds = model(inputs)
```

And this is how we do it!
Notice the use of model.eval() and torch.no_grad(). These are 2 techniques I explain in my Neural Optimization course, the main idea is to set the gradients to OFF so we can run faster!

Summary

Let's summarize what we've learned in this introduction to PyTorch.

- There are really 3 things important to understand, and that are different from other libraries. **Datasets & DataLoaders**, **Init & Forward**, **Train & Run.**

- The **Dataset** is a class you create with Python. Because it's a custom class, you need to implement the **init()** function, but also the **len()** or **getitem()** functions. When building a classic class, for example a string, these are defined already; but because this is custom, we need to redefine them.

- The **DataLoader** is the PyTorch way to encapsulate a Dataset and send it to a model.

- A model has 2 functions: **init()** and **forward(). Init()** is the definition of the operations we'll use while **forward()** is the actual recipe to call these operations in the right order.

- To **train a model**, you need to implement 5 steps:
    1. Loop through the DataLoader
    2. Collect our input
    3. Run a forward pass
    4. Calculate the loss
    5. Backpropagate

- To run a model, freeze the gradients and then run **model(inputs).**

- The minimal codes are here to help you, so please take a look at the next page!

Appendix: Minimal Codes

Dataset:

```
class BDD100k_dataset(class):
  def __init__(self):
    pass
  def __len__(self):
    pass
  def __getitem__(self):
    Pass
```

Model:

```
class ModelToBuild(nn.Module):
    def __init__(self, net):
        pass

    def forward(self, x):
        pass
```

Training:

```
for epoch in range(n_epochs):
    model.train() # Set model to training

    for i, data in enumerate(train_dataloader):
        input = ### GET INPUT FROM "DATA"
        optimizer.zero_grad() #NO GRADIENTS
        output = model(input) #INFERENCE
        loss_1 = ### GET LOSS
        loss.backward() #COMPUTES THE GRADIENTS DURING BACK PROPAGATION
        optimizer.step() # UPDATE THE WEIGHTS
```

Inference:

```
model.eval()
with torch.no_grad():
    inputs = inputs.to(device)
    y_preds = model(inputs)
```