CNN using Tensorflow

What are Tensors?

There are two types of tensors : i) one type is used by mathematicians and physicists. We do not talk about these today, or these are different from ML lingo tensors.

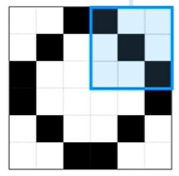
The other type of tensor is used in neural networks this is the type that **hold the data and the weights and biases** and **are designed for hardware acceleration** so that neural networks can do all the math they need to do in a relatively short period of time . They (tensors) also take care of back propagation with automatic differentiation

In ML lingo of Array v/s Tensors:

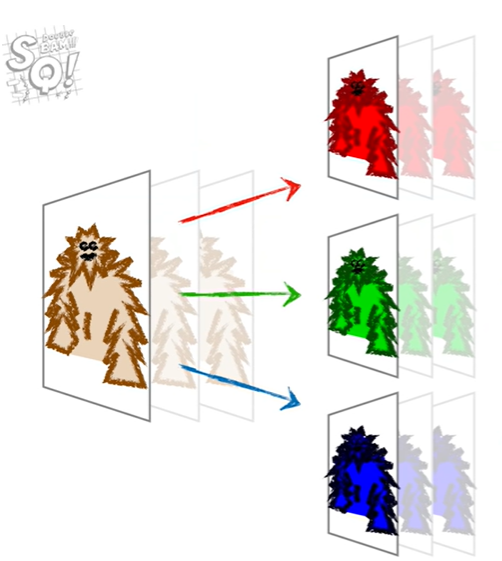
1. A single value as input: --- Scalar v/s 0-dimension Tensor
2. A series of values as input : ---- a 1d Array v/s 1d Tensor



1. A single 2d image(greyscale) as input:---- Matrix (or, 2D array) v/s 2d Tensor



1. A video as input : ----- nD array (or, Multi-dimension array) v/s nD tensor



In tensorflow the rank of a tensor is its dimensionality. Here are some examples:

---------------------------------------------------------------

| Rank of | Math | Example

| tensor | entity |

---------------------------------------------------------------

| 0 | Scalar | x = 42

| 1 | Vector | z = [10, 15, 20]

| 2 | Matrix | a = [[1 0 2 3],

| | | [2 1 0 4],

| | | [0 2 1 1]]

| 3 | 3-Tensor | A single image of shape:

| | | [height, width, color\_channels]

| | | ex: [1080, 1920, 3]

| 4 | 4-Tensor | A batch of images with shape:

| | | [batch\_size, height, width, channels]

| | | ex: [10, 1080, 1920, 3]

| N | n-dim | You get the idea...

| | Tensor |

---------------------------------------------------------------

**NOTE:**

**In many situations, certain hardwares called TPU (Tensor Processing Unit) are specifically designed to work with Tensors and majke N.N. run more quickly.**

Modules and methods in Tensorflow:

1. **Torch.nn.Module():**

import torch.nn as nn

class Model(nn.Module):

1. **training** ([*bool*](https://docs.python.org/3/library/functions.html#bool)) – Boolean represents whether this module is in training or evaluation mode.
2. parameters(*recurse=True)*

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Return type: [*Iterator*](https://docs.python.org/3/library/typing.html#typing.Iterator)[[*Parameter*](https://pytorch.org/docs/stable/generated/torch.nn.parameter.Parameter.html#torch.nn.parameter.Parameter)]

Example:

for param in model.parameters():

print(type(param), param.size())

output: <class 'torch.Tensor'> (20L,)

1. train(*mode=True*)

Sets the module in training mode.

Parameters:

**mode** ([*bool*](https://docs.python.org/3/library/functions.html#bool)) – **whether to set training mode (True) or evaluation mode (False). *Default value is True.***

Returns: self

Return type: Module

1. **Optimizer Class:**
2. To construct an Optimizer you have to give it an iterable containing the parameters (all should be Variable s) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.

Example:

**opt** **=** torch.**optim.SGD(model.parameters(),** **lr=**0.01**,** **momentum=**0.9**)**

**optimizer** **=** torch.**optim.Adam([var1,** **var2],** **lr=**0.0001**)**

1. zero\_grad(*set\_to\_none=False*)[[SOURCE]](https://pytorch.org/docs/stable/_modules/torch/nn/modules/module.html#Module.zero_grad)

Sets gradients of all model parameters to zero.

**This comes the *all important step* of Training:**

1. Zeroing our gradient : opt.zero\_grad(): Initialize the derievatives as 0, each time before passing a sample(or an image).
2. Performing backpropagation: loss.backward() : **In this step, we calculate the derivative of the loss w.r.t. parameters(i.e.w and b), and we use these derivatives in the next step.**
3. Updating the weights of our model : opt.step() :

**3. Datasets and DataLoaders**

Code for processing data samples can get messy and hard to maintain; we ideally want our dataset code to be decoupled from our model training code for better readability and modularity. PyTorch provides two data primitives: torch.utils.data.DataLoader and torch.utils.data.Dataset that allow you to use pre-loaded datasets as well as your own data. Dataset stores the samples and their corresponding labels, and DataLoader wraps an iterable around the Dataset to enable easy access to the samples.

Example:

**import** **torch**

**from** **torch.utils.data** **import** [**Dataset**](https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset)

**from** **torchvision** **import** **datasets**

**from** **torchvision.transforms** **import** [**ToTensor**](https://pytorch.org/vision/stable/generated/torchvision.transforms.ToTensor.html#torchvision.transforms.ToTensor)

**import** **matplotlib.pyplot** **as** **plt**

[**training\_data**](https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html#torchvision.datasets.FashionMNIST) **=** [**datasets.FashionMNIST**](https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html#torchvision.datasets.FashionMNIST)**(**

**root=**"data"**,**

**train=True,**

**download=True,**

**transform=[ToTensor](https://pytorch.org/vision/stable/generated/torchvision.transforms.ToTensor.html" \l "torchvision.transforms.ToTensor" \o "torchvision.transforms.ToTensor)()**

**)**

[**test\_data**](https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html#torchvision.datasets.FashionMNIST) **=** [**datasets.FashionMNIST**](https://pytorch.org/vision/stable/generated/torchvision.datasets.FashionMNIST.html#torchvision.datasets.FashionMNIST)**(**

**root=**"data"**,**

**train=False,**

**download=True,**

**transform=[ToTensor](https://pytorch.org/vision/stable/generated/torchvision.transforms.ToTensor.html" \l "torchvision.transforms.ToTensor" \o "torchvision.transforms.ToTensor)()**

**)**

**OR,OTHER WAYS TO IMPORT DATASET AND DATALOADER**

from torchvision.datasets import FashionMNIST()

from torch.utils.data import DataLoader

# NOW, load the KMNIST “**dataset”**

**# root (string) – Root directory of dataset where MNIST images exist.**

**# transform = () , The transform we need is to convert the NumPy array loaded by PyTorch into a tensor data type.**

print("[INFO] loading the KMNIST dataset...")

trainData = KMNIST(root="data", train=True, download=True,

transform=ToTensor())

testData = KMNIST(root="data", train=False, download=True,

transform=ToTensor())

# initialize the train, validation, and test “**data loaders”**

**#** We have loaded that dataset into the DataLoader and can iterate through the dataset as needed. Each iteration below returns a batch of train\_features and train\_labels (containing batch\_size=64 features and labels respectively). Because we specified shuffle=True, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order

# We set shuffle=True only for our trainDataLoader since our validation and testing sets do not require shuffling.

trainDataLoader = DataLoader(trainData, shuffle=True,

batch\_size=64)

valDataLoader = DataLoader(valData, batch\_size=64)

testDataLoader = DataLoader(testData, batch\_size=64)

1. What is difference between nn.Module and nn.Sequential

* I should start by mentioning that nn.Module is the base class for all neural network modules in PyTorch. As such nn.Sequential is actually a direct subclass of nn.Module, you can look for yourself on this line.
* When creating a new neural network using nn.Module, you would usually go about creating a new class and inheriting from nn.Module, and defining **two methods: \_\_init\_\_ (the initializer, where you define your layers) and forward (the inference code of your module, where you use your layers).** *That's all you need, since PyTorch will handle backward pass with Autograd.*

class NN(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.fc1 = nn.Linear(10, 4)

self.fc2 = nn.Linear(4, 2)

def forward(self, x)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

return x

BUT, If the model you are defining is sequential, i.e. the layers are called sequentially on the input, one by one. Then, you can simply use a nn.Sequential. **The objective of nn.Sequential is to quickly implement sequential modules such that you are not required to write the forward definition, it being implicitly known because the layers are sequentially called on the outputs**. As I explained earlier, nn.Sequential is a special kind of nn.Module made for this particular widespread type of neural network. The equivalent here is:

class NN(nn.Sequential):

def \_\_init\_\_(self):

super().\_\_init\_\_(

nn.Linear(10, 4),

nn.ReLU(),

nn.Linear(4, 2),

nn.ReLU())

*Or a simpler way of putting it is:*

NN = Sequential(

nn.Linear(10, 4),

nn.ReLU(),

nn.Linear(4, 2),

nn.Linear())

1. **Reshaping the Tensor**

**Method 1 : Using reshape() Method**

This method is used to reshape the given tensor into a given shape( Change the dimensions)

***Syntax:****tensor.reshape([row,column])*

***where,***

* *tensor is the input tensor*
* *row represents the number of rows in the reshaped tensor*
* *column represents the number of columns  in the reshaped tensor*

**Example 1:**Python program to reshape a 1 D tensor to a two-dimensional tensor.

* Python3

|  |
| --- |
| # import torch module  **import** torch    # create an 1 D etnsor with 8 elements  a **=** torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])    # display tensor shape  **print**(a.shape)    # display  actual tensor  **print**(a)    # reshape tensor into 4 rows and 2 columns  **print**(a.reshape([4, 2]))    # display shape of reshaped tensor  print(a.shape) |

**Output:**

torch.Size([8])

tensor([1, 2, 3, 4, 5, 6, 7, 8])

tensor([[1, 2],

[3, 4],

[5, 6],

[7, 8]])

torch.Size([8])

**Method 2 : Using flatten() method**

flatten() is used to flatten an N-Dimensional tensor to a 1D Tensor.

***Syntax:****torch.flatten(tensor)*

*Where, tensor is the input tensor*

**Example 1:**Python code to create a tensor  with 2 D elements and flatten this vector

|  |
| --- |
| # import torch module  **import** torch    # create an 2 D tensor with 8 elements each  a **=** torch.tensor([[1,2,3,4,5,6,7,8],                    [1,2,3,4,5,6,7,8]])    # display actual tensor  print(a)    # flatten a tensor with flatten() function  print(torch.flatten(a)) |

* **Output:**
* tensor([[1, 2, 3, 4, 5, 6, 7, 8],
* [1, 2, 3, 4, 5, 6, 7, 8]])
* tensor([1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8])

## ****Method 3: Using resize() method****

This is used to resize the dimensions of the given tensor.

***Syntax:****tensor.resize\_(no\_of\_tensors,no\_of\_rows,no\_of\_columns)*

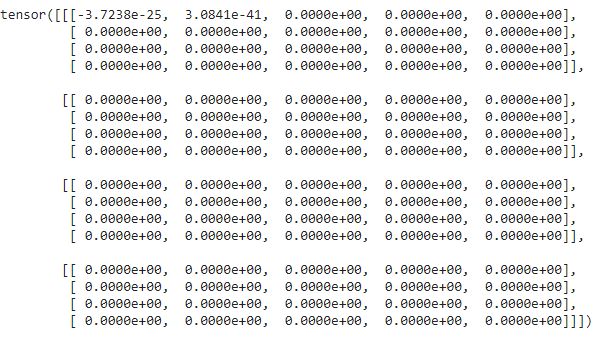
***where:***

* *tensor is the input tensor*
* *no\_of\_tensors represents the total number of tensors to be generated*
* *no\_of\_rows represents the total number of rows in the new resized tensor*
* *no\_of\_columns represents the total number of columns in the new resized tensor*

**Example 1:**Python code to create an empty one D tensor and create 4 new tensors with 4 rows and 5 columns.

|  |
| --- |
| # importing torch module  **import** torch    # create one dimensional tensor  a **=** torch.Tensor()    # resize the tensor to 4 tensors.  # each tensor with 4 rows and 5 columns  print(a.resize\_(4, 4, 5)) |

**Output:**



1. **Use of ‘model.eval()’ and ‘with torch.no\_grad()’ in PyTorch model evaluate**

The process of running a trained model on new data is called evaluation or inference in deep learning. In order to do an evaluation, we need to put the network in eval mode:

model.eval()

There are certain layers that behave differently in training and evaluation modes, such as the **dropout and batch normalize layers**.

**Dropout layer with eval mode**: nodes are only randomly dropped during training, not evaluation or inference.

**Batch normalization** : the behavior for updating the learnable parameters depends on whether the model is a training model or not. These parameters are learned only during training and are then used for normalization during the evaluation

*Why is it called batch normalization?*

In order to bring all the activation values to the same scale, we normalize the activation values such that the hidden representation doesn’t vary drastically and also helps us to get improvement in the training speed.

Since we are computing the mean and standard deviation from a single batch as opposed to computing it from the entire data

EXAMPLE:

|  |
| --- |
|  |
|  | class MyNetBN(nn.Module):  def \_\_init\_\_(self): |
|  | super(MyNetBN, self).\_\_init\_\_() |
|  | self.classifier = nn.Sequential( |
|  | nn.Linear(784, 48), |
|  | nn.BatchNorm1d(48),  #applying batch norm |
|  | nn.ReLU(), |
|  | nn.Linear(48, 24), |
|  | nn.BatchNorm1d(24), |
|  | nn.ReLU(), |
|  | nn.Linear(24, 10) |
|  | ) |
|  |  |
|  | def forward(self, x): |
|  | x = x.view(x.size(0), -1) |
|  | x = self.classifier(x) |
|  | return x |

## with torch.no\_grad():

Here with this statement , we are telling that we do not want gradients as we will not want to update the parameters.

1. **tensor.detach().cpu().numpy():** It isUsed to convert a torch.Tensor (on GPU) to a numpy.ndarray (on CPU)

**Tensor.detach()** is used to detach a tensor from the current computational graph. It returns a new tensor that doesn't require a gradient.

* When we don't need a tensor to be traced for the gradient computation, we detach the tensor from the current computational graph.
* We also need to detach a tensor when we need to move the tensor from GPU to CPU.(mostly used for this).

### **Syntax**

Tensor.detach()

It returns a new tensor **without** **requires\_grad = True**. Or, with requires\_grad = False.

**Tensor.cpu() :** Used to move a torch tensor from GPU to CPU, the following syntax/es are used −

Syntax:

Tensor.to("cpu")

And,

Tensor.cpu()

**Tensor.cuda():** To move a torch tensor from CPU to GPU, following syntax/es are used −

Syntax:

Tensor.to("cuda:0")

or

Tensor.to(cuda)

And,

Tensor.cuda()

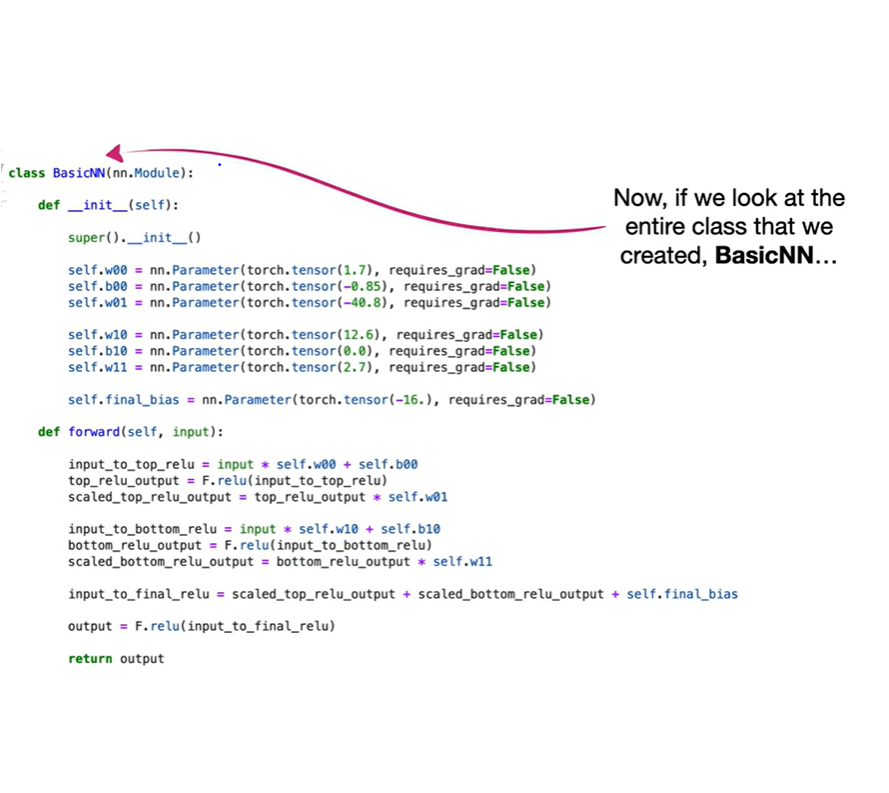
**Example:**  I have a CUDA variable, called “var”, that I want to read out its value into numpy (say for plotting).

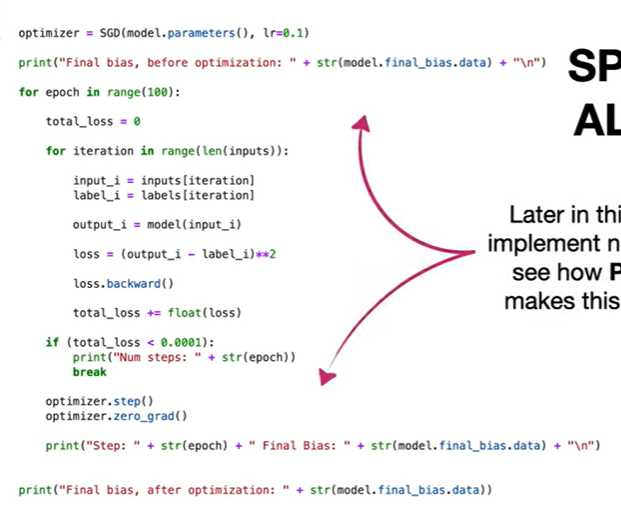
Try 1: If I do var.numpy() I get RuntimeError: Can’t call numpy() on Variable that requires grad. Use var.detach().numpy() instead.

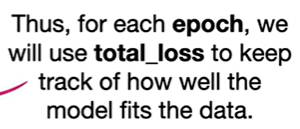
Try 2: Ok, so I do var.detach().numpy() and get TypeError: can’t convert CUDA tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first

Try 3: Ok, so I go var.detach().cpu().numpy() and it works.

StatQuest Example:







First we use loss dot backward to

calculate the derivative of the loss

function with respect to the parameter

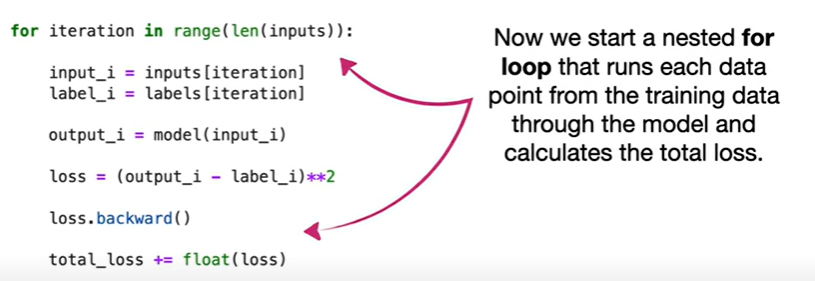
or parameters we want to optimize

**in this example, that means calculating**

**the derivative of the squared residual**

**with respect to b sub final** and plugging

in the predicted and known values



in this case that means the for loop

starts with the first point in the

training data

and determines its input or dose and its

known label or effectiveness

then it runs that dose through the model

to get a predicted output

and then we calculate the loss between

the predicted value and the known label

with a loss function

in this case we are calculating the

squared residual where the residual is

the difference between the output and

the known value

that said you can code any loss function

that you want to use like the absolute

value loss

or you can choose from among the many

loss functions like mse loss or cross

entropy loss that come with pi torch