PyTorch – freeCodeCamp – Basics of Pytorch – tensors and gradients

Difference between classical programming and machine learning

In classical programming, you set the rules and when data arrives, the rules give you answers. E.g – if you need to find the shortest path, you must work out bfs or dfs and come up with answers.

In case of machine learning, you are given data and answers and the task it to figure out the rules. E.g – you are given two images of cat and dog and say that which is which. Now form a set of images of cats and dogs your algorithm must be able to discern cats and dogs.

What is tensor?

Tensor can be anything, from a number, to an array, to a vector, to a vector of vector of vectors aka n-dimensional array.

Matrix multiplication along with other operations such as addition, division, etc. can be done easily using tensors.

Tensor maintains *conformity*. If in the 4th dimension a row has 12 columns, all the rows will have equal number of columns. If one of those columns have 7 rows than all 12 columns will have 7 rows.

The term gradients and derivatives are same. Its mathematical expression is dy/dx and it is the slope of a curve. ***Gradient or derivative is for 1 unit of movement along with x-axis, how much movement happens in y-axis.***

Derivatives are used when you are dealing with numbers and gradients are used when you are dealing with matrices.

PyTorch Functions

Basics: Pytorch is popular for two reasons:

1. Imperative programming – calculations happen with the flow of the code – it means that when you run the code it compiles and runs at the same time (NOT CLEAR)
2. Dynamic Computational Graph – built at runtime – graph structure can be changed according to the input data – debugging is easy

# **Linear Regression**

import torch.nn as nn

torch.from\_numpy() [takes in numpy matrix and turns into a tensor. Why? Numpy works best for CPUs; whereas, tensors are built for parallel computation using GPUs.]

from torch.utils.data import TensorDataset [Helps to create batch from datasets]

train\_ds = TensorDataset(inputs, targets)

from torch.utils.data import DataLoader [splits data into pre-defined size while training and helps to shuffle, random sampling of data]

batch\_size = 5

train\_dl = DataLoader(train\_ds, batch\_size, shuffle=True)

model = nn.Linear(3,2) [#of inputs and outputs]

import torch.nn.functional as F

loss\_func = F.mse\_loss

loss = loss\_func(model(inputs), targets)

if you put “**?F.mse\_loss”** inside a cell, it will show details about the Mean Square Function. Similar goes for everything.

opt = torch.optim.SGD(model.parameters(), lr=le-5) // lr stands for learning rate

def fit(num\_epochs, model, loss\_func, opt):

for epoch in range(num\_epochs):

for xb, yb in train\_dl:

pred = model(xb) // generating predictions

loss = loss\_func(pred, yb) // comparing predictions with outputs

loss.backward() // computing the gradients

opt.step() // NEW :- update parameters with gradients

opt.zero\_grad() // resetting the gradients to zero

if (epoch+1) % 10 == 0

print(‘Epoch [{}/{}], Loss: {: .4f}’ .format(epoch+1, num\_epochs, loss.item()))

now to train the model:

fit(100, model, loss\_func, opt)

Summary of Linear Regression & Gradient Descent

* Linear Regression is simply weighted parameters plus added bias that leads to prediction closer to the actual output.

Wi = weight ; Pi = parameter ; B = Bias; Y = output

W1\* P1 + W2\* P2 + W3\* P3 + W4\* P5 + B = Y(pred)

Loss = Y(actual) – Y (pred)

* Gradient/derivative of loss with respect to weights & biases is calculated using – loss.backward()
* Gradients are stored in .grad property of the respective tensors.
  + W.grad
* Gradient is positive means that the slope is positive, meaning from left to right the slope is rising
  + If we increase the value of the weight element (value on x-axis) the loss will increase
  + If we decrease the value of the weight element the loss will decrease
* Gradient is negative means that the slope is decreasing from left to right
  + If we drag the value of weight element to the right (increase), the loss will decrease
  + If we drag the value of the weight element to the left (decrease), the loss will increase
* Automatic way to manipulate the model’s weights and biases using gradients is through optimizers
  + torch.optim.SGD(model.parameters(), lr=1e-5)/ torch.optim.Adam(model.parameters(), lr=1e-5)
* Adsf
* \asdf
* Sdf

PyTorch & Torchvision on MNIST handwritten Dataset

Images are in PIL format and needed to be converted into tensors. If there is no function that can be as derivation than gradient descent cannot be used.

Images are an object of the class *PIL.Image.Image*, which is a part of the python imaging library **Pillow**.

%matplotlib inline helps to show the image in jupyter notebook, otherwise it will be shown as a popup.

Following are the steps to change data images into inputs

1. We must convert images into tensors. *torchvision.transforms*
2. To convert from image to tensors -> *torchvision.transforms.ToTensor()*
3. Next is importing DataLoader from torch.utils.data

Loss Function:

Cross Entropy: to calculate the total entropy between two distributions.

**Cross­\_Entropy : -** *SUM [ Y(actual) \* log (Y(pred));*

*Using CE we can make the state differentiable.*

For any classification problem, categorical cross entropy is the preferred method ( as a loss calculator).

Extending the *nn.module* class :

SoftMAX : converts outputs into positives and then to probabilities. torch.nn.functional.softmax()

Output : y(hat) -> e^(y(hat)) / SUM e^(y(hat))

We will train the model on a batch and validate it on a batch in each epoch:

1. For training a batch this are the steps:
   1. Generate predictions
   2. Calculate loss
   3. Compute Gradients
   4. Update weights
   5. Reset gradients
2. For validating a batch
   1. Generate prediction
   2. Calculate loss
   3. Calculate metrics

##################################################################

def fit(epochs, lr, model, train\_loader, val\_loader, opt\_func=torch.optim.SGD):

optimizer = opt\_func(model.parameters(), lr)

history = [] # for recording epoch-wise results

for epoch in range(epochs):

# Training Phase

for batch in train\_loader:

loss = model.training\_step(batch)

# the following computes gradients – takes derivatives of loss w.r.t weights # and biases

loss.backward()

optimizer.step() # this is the gradient dissent step – updates the parameters

optimizer.zero\_grad() # resets the gradient

# Validation phase

result = evaluate(model, val\_loader)

model.epoch\_end(epoch, result)

history.append(result)

return history

##################################################################

Activation Function: Inserts non-linearity to the hidden layer. ReLU for instance, pick maximum between 0 and X. ReLU -> max(0, x)

Image Classification using CNNs

CIFAR10 Dataset – 60,000 32 x 32 color images and 10 output classes – CNN

**Import large Google Drive files directly into your Kernel**

Go to the settings of your Kernel.

Turn on the internet.

Then, install this particular module.

pip install gdown

Now, all you need is the URL of the google drive file you want to import.

Also, be mindful of the file extension. The file in my example was a binary file saved using numpy (.npy extension). The extension of the output file should be kept same as the extension of the original file.

import gdown

url = 'https://drive.google.com/uc?id=1B6\_rtcmGRy49hqpwoJT-\_Ujnt6cYj5Ba'

output = 'file.npy'

gdown.download(url, output, quiet=False)

You will fine the file in the working directory of your kernel @

/kaggle/working/file.npy

Why is it required to move the channel at the end?

* Pytorch puts the channel at the beginning – if you give that tensor to matplotlib it throughs an error

**Idea of Convolutional Network:**

If you have an image of size 32 x 32 and you take a kernel of 3 x 3 and slide the kernel from the top left corner of the image and do element wise multiplication you will get a resultant matrix.

Padding helps you not lose the dimensions after a convolution operation

Stride tells you to the number of pixels space you should move

<https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>

<https://sgugger.github.io/convolution-in-depth.html>

In a fully connected layer the input and outputs are multiplied to calculate the number of parameters. It will result in every output feature being the weighted sum of every single input feature.

*On the contrary, Convolutions allow us to do this transformation with each output feature, instead of “looking at” every input feature, only getting to “look” at input features coming from roughly the same location.*

Padding and Strides are two commonplace techniques in Convolution Layers.

When the input image has only 1-channel, Filter and Kernel are indistinguishable. Whereas, in case of input image with 3-channels, Filer and Kernel are very different.

Filter is a collection of kernels. Each channel will have a kernel of its own and each of those kernels can be unique.

Each filter in a convolution layer produces one and only one output channel. How? The filter takes an image with 3-channels, uses 3 kernels to get the 3-channels, sums those 3-channels and outputs 1-channel. Finally, a bias term is added to that to that kernel and output a final result.

So how many channels we get at the end of a layer? – ***the number of filters used on the previous layer.***

(5 x 5 x 3) (3x3 x 10) = 3 x 3 x 10

(Image\_shape x Channels) (Kernel x Filter) = (Output\_Matrix\_shape x channels)

Output\_Matrix\_shape = Image\_shape – kernel + 1

Input\_image\_shape = 32 x 32

Padding = 1

Input\_image\_shape = 34 x 34

Kernel = 3 x 3

After convolution –

Output\_Matrix\_shape = 34 – 3 + 1 x 34 – 3 + 1 = 32 x 32

Hence, padding with 1 helps to retain the same output size.

The Idea of Receptive Field:

CNN’s architecture was designed in a way that allows the input size to grow smaller but the number of channels grow deeper when moving from the input layer to the final layer.

To make it smaller, strides or pooling layers are used. The receptive field determines what area of the original input to the entire network the output gets to see.

*The CNN, with the priors imposed on it, starts by learning very low level feature detectors, and as across the layers as its receptive field is expanded, learns to combine those low-level features into progressively higher level features; not an abstract combination of every single pixel, but rather, a strong*visual hierarchy *of concepts.*

*By detecting low level features, and using them to detect higher level features as it progresses up its visual hierarchy, it is eventually able to detect entire visual concepts such as faces, birds, trees, etc, and that’s what makes them such powerful, yet efficient with image data.*

***Robustness against adversarial attacks is currently a highly active area of research, the subject of many papers and even competitions, and solutions will certainly improve CNN architectures to become safer and more reliable.***

Following LINKs: <https://cs231n.github.io/convolutional-networks/>

<https://distill.pub/2017/feature-visualization/>

<https://openai.com/blog/adversarial-example-research/>

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

Overfitting: After training for a while, if a trend arrives where the training loss is decreasing but the validation loss is either increasing or flattened out, it is because of overfitting.

Residual Networks, Data Augmentation, and Regularization (dropout)

Let’s begin...

**Channel-wise data normalization:**

Image tensors are normalized by subtracting the Mean and dividing by the Standard Deviation across each channel. The mean becomes 0 and standard deviation becomes 1 for the data across each channel (not sure I understand it fully but I’ll allow it).

***Why normalize?*** – Normalizing the data prevents the values from any one channel from disproportionately affecting the losses and gradients while training. Certain values affect the losses and gradients when they have a higher or wider range of values than others.

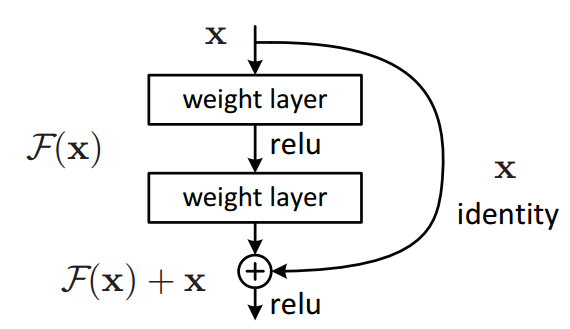
**Data Augmentation:**

Taking a picture, adding a padding of 4 and then randomly crop and flip the image horizontally will make the model see a slightly different image in every training epoch.

**Residual Blocks:**

**Link:** [**https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec**](https://towardsdatascience.com/residual-blocks-building-blocks-of-resnet-fd90ca15d6ec)

*Understanding a residual block is quite easy. In traditional neural networks, each layer feeds into the next layer. In a network with residual blocks, each layer feeds into the next layer and directly into the layers about 2–3 hops away.*



**Vanishing Gradients and Curse of Dimensionality:**

**Degradation problem:**

If we keep increasing the number of layers, we will see that the accuracy will saturate at one point and eventually degrade. And, it is usually not caused due to overfitting. A counter-intuitive problem arises where the shallower networks learn better than their deeper counterpart. But this is what is seen in practice and is popularly known as the degradation problem.

Identity function: where x == y; the graph is a diagonal from left to right going UP through the origin. 2x + 3x = 5x is an identity function.

*You can skip the training of few layers using skip connections or residual connections. This is what we see in the image above. In fact, if you look closely, we can directly learn an identity function by relying on skip connections only. This is the exact reason why skip connections are also called identity shortcut connections too. One solution for all the problems!*

*Let us consider a neural network block whose input is x, and we would like to learn the true distribution H(x). Let us denote the difference or the residual between this as:*

*R(x) = Output – Input = H(x) – x*

* *H(x) = R(x) + x*

*So, to get the true distribution, you need to add the residue to the input.*

**Batch Normalization and Dropout in NN:**

**Link:** [**https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd**](https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd)

If different types of input values range verily inside different ranges, it can cause problems in backpropagation.

|  |  |
| --- | --- |
| X | Y |
| -300 | 2 |
| 37 | 7 |
| -248 | 9 |
| 199 | 0 |
| 300 | 4 |
| 40 | 10 |

X ranges from [-300, 300] but Y ranges from [0, +10];

Now weights associated with the values of X will have higher impact than weighs associated with the values of Y. When you backpropagate (compute gradients, do gradient descent, and make gradients zero), weights associated with X and Y will affect differently.

Un-normalized data will lead to oscillation in the plateau area before finding the global minima.