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

# **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.

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

*Using CE we can make the state differentiable.*

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)

loss.backward()

optimizer.step()

optimizer.zero\_grad()

# 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)