Intro to ML - CS 419

Lecture 7: Tools Used in ML Implementation

3rd February 2023

Lecturer: Abir De *Scribe: Groups 13 & 14*

This lecture was a tutorial on numpy, matplotlib, pytorch and torchviz. The main goal of the tutorial was to give an introduction to pytorch and also to explain why vectorization is important in machine learning.

1 Vectorization of code

ML models generally need to process a large volumne of data. Thus if the algorithm being used is slow then it won't be applicable for any practical problem. It would only exist in theory. Vectorization keeps code computationally efficient.

Python has a few libraries like numpy and pytorch that can be used for executing operation over arrays without having to write loops.

```
num_items = 7
dim = 50
A_np = np.random.rand(num_items,dim)
B_np = np.random.rand(num_items,dim)

A_to = torch.rand(num_items,dim)
B_to = torch.rand(num_items,dim)
```

The above code is used for generating 2 numpy matrices A_np and B_np. These have a dimension of 7×50. Random numbers from a uniform distribution over $[0, 1)$ are drawn to fill these matrices.
Similarly the second half of the code does the same thing, except the a torch Tensor is formed in this case.

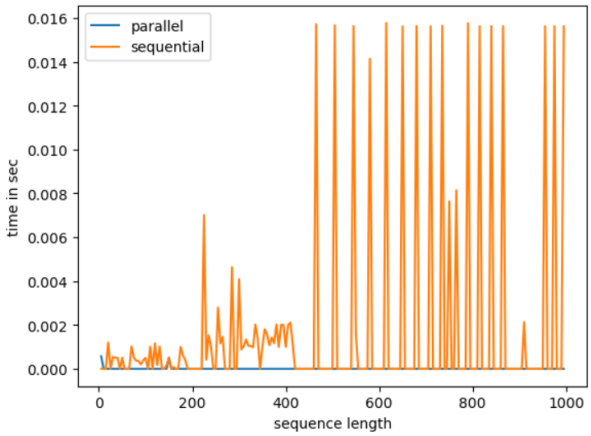
```
def dot_prod_novec(A,B):
  s = time.time()
  op_list = []
  for idx in range(len(A)):
    op_list.append(sum(A[idx]*B[idx]))

  return (np.array(op_list), time.time()-s)


def dot_prod_vec(A,B):
  s = time.time()
  output = (A*B).sum(-1)
  return (output, time.time()-s)
```

Here two functions are described for finding the dot product of rows in 2 matrices. dot_prod_novec finds the dot products by iterating over rows in matrix A and multiplying each element with the corresponding element in B and then taking a sum. This is the sequential code
dot_prod_vec uses numpy for multiplying the matrices element wise and then taking sum over columns. This is the vectorized code
The first function uses a for loop for finding the dot product of each row in A and B. Here computations happen sequentially. The second function uses numpy which can divide tasks into subtasks and process them parallely.
The time taken for running each function was plotted as a function of matrix size. From Figure 1 it is clear that the vectorized code was much more efficient than the sequential code.

Figure 1: Comparing sequential vs vectorized code for numpy matrices

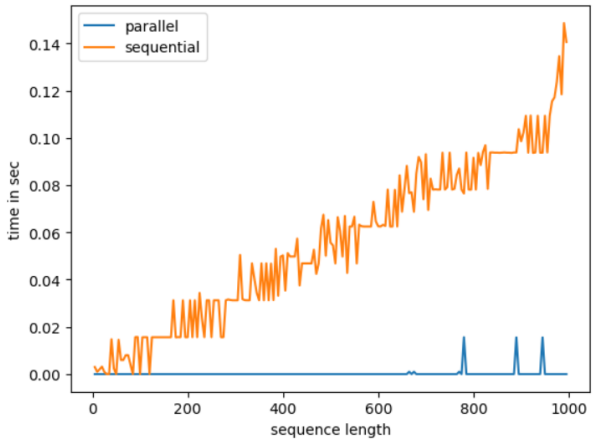


Thus it is clear that the vectorized code was significantly faster than sequenctial code.

```
time_parallel = []
time_seq = []
for num_items in range(5,1000,5):
A = torch.rand(num_items,dim)
B = torch.rand(num_items,dim)
op_vec, time_vec = dot_prod_vec(A,B)
op_novec, time_novec = dot_prod_novec(A,B)
#assert (op_vec == op_novec).all()
assert np.allclose(op_vec,op_novec)
time_parallel.append(time_vec)
time_seq.append(time_novec)
```

The above code compares the dot_prod_vec function and the dot_prod_novec function for torch tensors. Here too the vectorized code is much more efficient than the sequential code as seen in Figure 2.

Figure 2: Comparing sequential vs vectorized code for torch tensors



Another example was presented in the tutorial

```
def all_pair_hinge_diff_max(A,B):
  s = time.time()
  maxVal=0
  for a in A:
    for b in B:
      val = max(0,a-b)
      maxVal = max(maxVal, val)
  return maxVal, time.time()-s


def all_pair_hinge_diff_max_vec(A,B):
  s = time.time()
  maxVal = np.max(np.maximum(0,A[:,None] - B[None,:]))
  return maxVal, time.time()-s
```

The above 2 python functions mathematically represent

$$\max(\max_{i,j}(A_i - B_j), 0)$$

Here A_i and B_j are two arrays.
The first function all_pair_hinge_diff_max iterates over each element in the two arrays and finds the difference between them and stores it in the variable "val", if the difference is positive. If it is negative "val" is set to 0. It then updates the current maximum value which is stored in the variable "maxval" by taking the maximum of "val" and "maxval". This is the sequential code
The second function all_pair_hinge_diff_max_vec does the same but in a different way. A[:, None] adds another dimension to A. If A originally was a vector with dimension n A[:, None] will be a $n \times 1$ matrix. Similarly A[None, :] will be a $1 \times n$ matrix.
A peculiar step in the second function is the difference between A[:, None] and B[None, :]. This is a difference between a $n \times 1$ matrix and a $1 \times n$ matrix. numpy alloys this and calculates the difference using broadcasting.

With the help of broadcasting, we can perform arithmetic operations on arrays of different shapes. The arrays are stretched such that the have the same shape and the arithmetic operations are performed between the elements simultaneously. As an example take two arrays $A = [1, 2, 3]$ and $B = [3, 4, 5]^T$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix}$$

After stretching the matrices

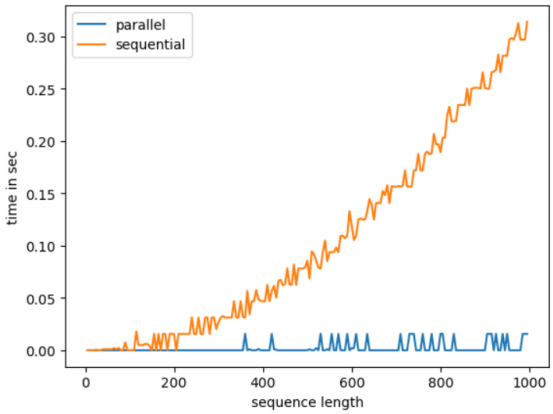
$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 3 & 3 & 3 \\ 4 & 4 & 4 \\ 5 & 5 & 5 \end{bmatrix}$$

And now after performing element wise addition

$$\begin{bmatrix} 4 & 5 & 6 \\ 5 & 6 & 7 \\ 6 & 7 & 8 \end{bmatrix}$$

The same principle is used for subtracting A and B in all_pair_hinge_diff_max_vec. Here due to broadcasting the operations take place simultaneously thus this code is vectorized.
The time taken for executing the ocde as a function of array size was plotted.

Figure 3: Comparing all_pair_hinge_diff_max and all_pair_hinge_diff_max_vec



Thus here too the vectorized code performs much better than the sequential code.

2 Plotting through Matplotlib

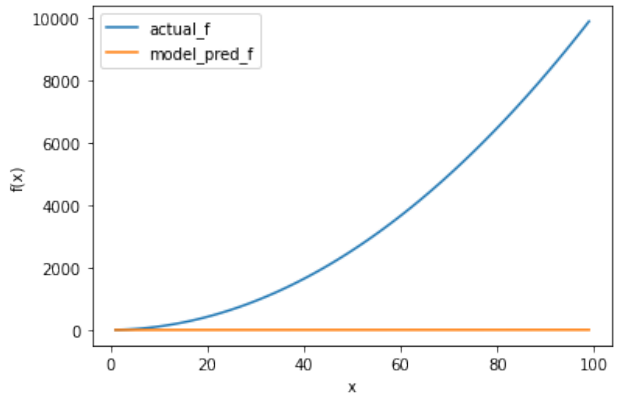
Matplotlib is the plotting library for creating static, animated, and interactive visualizations in Python.

```
from matplotlib import pyplot as plt
```

Pyplot is a Matplotlib module that provides a MATLAB-like interface. Matplotlib is designed to be as usable as MATLAB, with the ability to use Python and the advantage of being free and open-source.

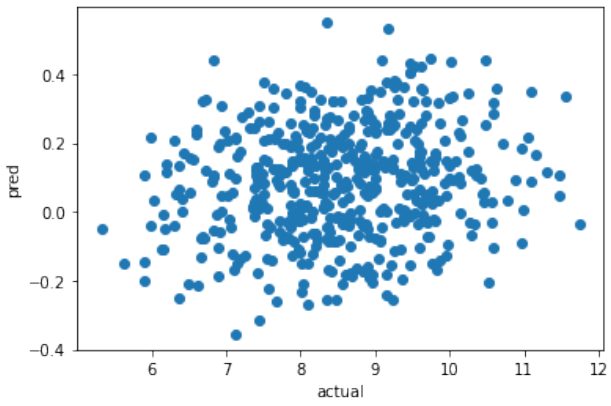
```
plt.plot(X,Y, label='actual_f')
plt.plot(X, model(X).detach(), label='model_pred_f')
plt.ylabel("f(x)", fontsize=10)
plt.xlabel("x", fontsize=10)
plt.legend()
plt.show()
```

In this code, we have defined two plots with torch tensors defined in the argument of the plot() and then labeled them. Then the name of the x-axis and y-axis has been defined with font size equal to 10.'plt.legend()' generates box around the labeling of plots.'plt.show()' executes to generate the plot.




```
plt.scatter(Y, model(X).detach())
plt.ylabel("pred", fontsize=10)
plt.xlabel("actual", fontsize=10)
plt.show()
```

The torch tensors defined in the argument of scatter() at the beginning of the code are used to generate points in this 2-D plane.Then labeling is done along the horizontal and vertical axis.



3 Numpy basics

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays.

```
import numpy as np
a= np.array([[1,2,3,4,5.0],[5.0,6.0,7.8,4,8]],dtype='int32')
#shape referring no. of rows and columns
a.shape
```

Here the shape of the 2-D array'a' , which e get through this code is (2,5).

```
a=np.random.randint(2,9,size=(3,3))
print(a)
```

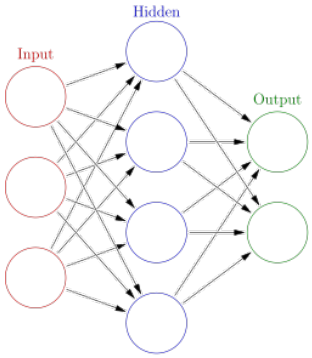
Through the above line of code, we are generating the array of size (3,3) with entries having random integer values starting from 2 to 8.

4 Neural Networks

A neural network processes data using generalizations and inferences. It utilises multiple layer structures, which can be summarised as the input, hidden and output layer, bias and linear layer specific layers, our focus is on the linear layer and what it does.
A use of the linear layer can be to learn the average rate of correlation between the output and the input variables, for example, to learn the constant 'w' when the relation between variables is y=w*x ; if x and y are positively correlate

=> w will be positive, if x and y are negatively correlated => w will be negative. If x and y are totally independent => w will be around 0.

Figure 4: Neural Network



5 PyTorch

The introduction of frameworks like PyTorch and Tensor Flow revolutionized the world of Data Science in around 2015.These are basically math libraries which help in implementing ML models. PyTorch is an open source ML framework which is based on Python and Torch library which was developed by Facebook's AI, it is mostly used by data scientists for research in AI and ML it is also used for image processing and Natural Language Processing(NLP).

5.1 Implementation of ML model using Torch

- **Torch Tensor**:
  A PyTorch Tensor is just like a numpy ndarray,the differnce being it runs both on CPU and GPU. We can even create a tensor from a numpy array by the following code

```
x_train_tensor = torch.from_numpy(x_train).float().to(device)
```

  Here x_train which was a pre initialized numpy array gets converted to torch tensor,the .float() function casts it to 32 bit float. When a tensor is created its stored in the CPU thus its a CPU tensor you can store it into GPU by using the .to(device) function where device is your GPU commonly referred as cuda or cuda:0.
  We can also convert our tensor back to numpy array, but first make sure to convert GPU tensors to CPU(you can check the type of tensor by using the type() command) by using Tensor.cpu() command.

- **Parameter Initialization**:
  Parameters are different from normal data in the sense that we need to compute gradients for the tensors and keep updating them. In PyTorch we can use this by using the requires_grad function

```
x = torch.tensor([[1.0, 3.0],
                  [2.0, -1.0],
                  [0.0, 1.0]], requires_grad=True)
```

Here we create a torch tensor x with requires_grad=True , if we want to assign our tensor to GPU we have to initialize them properly so that the requires_grad does not get masked. This function computes gradients on it's own by using the autograd tool of PyTorch.

- **Autograd**:
  Autograd is the automatic differentiation package of PyTorch with the help of autograd we don't need to go through gradient computation, chain rule etc. but it does all the stuff on its own. We can also check the computed gradient values by using grad() function.
  The idea is simple initialize the parameters with requires_grad, do the forward prop then use the loss.backward() function(loss computation is usually the last step of forward prop) update the parameters with the computed gradients, but there is a slight catch here. The grad computed here are accumulated so after updating you need to clear the gradient for which you can use the zero_() function at the end of the updating step. We should perform the update step such that it does not get added to the Computation graph because we don't want to use that step in our backprop so we use torch.no_grad for it

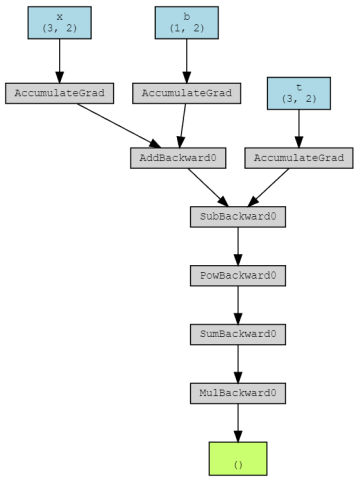
```
    with torch.no_grad():
    a -= lr * a.grad
    b -= lr * b.grad
```

- **Dynamic Computation Graph**:
  We can use the PyTorchViz package and use its make_dot method to visualize graphs on a tensors for the operations done on them. This graph has basically 3 components : blue, gray and green boxes. Blue boxes are our parameter tensors the gray boxes are the operations done on them or other dependencies, green boxes are just like gray ones except that they are the point from where backprop starts.

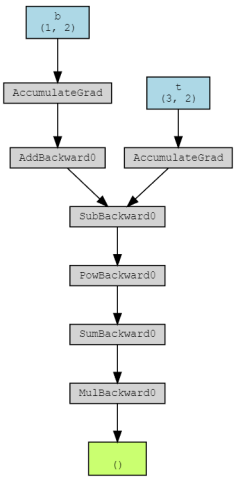
```
x = torch.tensor([[1.0, 3.0],
                  [2.0, -1.0],
                  [0.0, 1.0]], requires_grad=True)
b = torch.tensor([[2.0, 1.0]], requires_grad=True)
y = x + b
t = torch.tensor([[2.0, 4.0],
                  [4.0, 1.0],
                  [1.0, 1.0]], requires_grad=True)

error = 0.5 * torch.sum((y-t)**2)
print(error)
make_dot(error,params={'x':x, 'b':b, 't':t})
```

The above code initializes 3 parameters x,b and t and does some basic operations on them now we use the make_dot function, Graph 1 shows us the computation graph for this code now we change the code a little bit and set requires_grad=False for x we see a change in the computation graph Graph 2, x no longer appears in the graph as gradients are no longer required for it.



(a) Graph 1



(b) Graph 2

We can also detach the computation graph using .detach() function, this returns the tensor without the requires_grad=True and the gradients of these tensors will now not be calculated.

- **Loss Functions**:
  Torch also provides us with many loss functions out of which we use torch.nn.MSELoss() which uses the Mean Squared Error loss we can add all the losses or average them using the reduction argument, reduction = 'mean' averages them up reduction = 'sum' adds them.

- **Optimizer**:
  If we have a few number of parameters we can manually update them by using the gradient and settiing grad as zero_ at the end of each iteration, but this is not possible if we have many parameters so we use optimizers(such as SGD or Adam) .step() function provided by PyTorch.
  We a don't need to zero the gradients now we just invoke the zero_grad() method

```
loss_fn = torch.nn.MSELoss()
opt = torch.optim.SGD(model.parameters(), lr = 0.01)
epoch_loss= []
for epoch in range(5):
  loss = loss_fn(model(X), Y)
  opt.zero_grad()
  loss.backward()
  opt.step()
  epoch_loss.append(loss.item())
  if epoch%1==0:
    print(epoch)
    plt.plot(X,Y,label='actual_f')
    plt.plot(X,model(X).detach(),label='model_pred_f')
    plt.legend()
    plt.show()
```

The above code plots the output of the actual function w.r.t the model function.As we can see here we do not run the update steps manually rather the Optimizer updates it manually using the opt.step() function also, we have initialized the opt.zero_grad()

- **Models**:
  PyTorch provides us with nested Models such as Linear; it applies a linear transformation to the incoming data

$$y = W^T x + b \tag{1}$$

  We need to specify the number of input and the number of output features for it and also the bias; the snippet below shows an example

```
a = torch.rand((1,5))
op = linear1(a)
make_dot(op)
```

  We can also initialize our own parameters and apply linear model on them

```
my_weight_param = torch.nn.Parameter(torch.zeros(5,2))
my_bias_param   = torch.nn.Parameter(torch.ones(2))
print(f"Initialized weight paramter {my_weight_param}")
print(f"Initialized weight paramter {my_bias_param}")
a = torch.rand((1,5))
op = a@my_weight_param + b
make_dot(op)
```

  torch.zeros() initializes tensor with all zero elements, torch.ones() initialize tensor with all elements as one.
  We can also define a class which returns with a linear model we can do it by the snippet below:

```
  class LinReg(torch.nn.Module):
  def __init__(self):
    super(LinReg, self).__init__()
    self.lin = torch.nn.Linear(1, 1)
  def forward(self, x):
    return self.lin(x)
```

6 References

- https://pytorch.org/docs/stable/torch.html
- https://towardsdatascience.com
- https://matplotlib.org
- https://www.geeksforgeeks.org
- https://stackoverflow.com