

CPEG 592 – Assignment #2 - 2024

Programming Concepts for NLP and Transformer Implementation

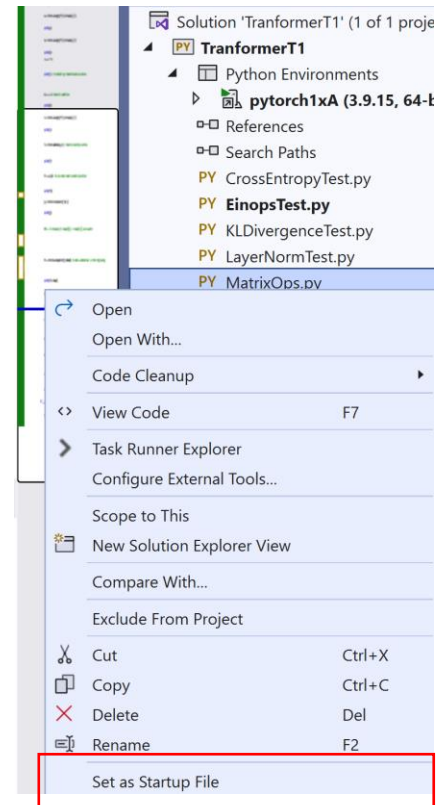
Before we program a transformer for NLP, we will review some of the important Pytorch programming concepts. Create a Python application called TransformerT1. Add a Python module to it called MatrixOps.py. Then type the following code in it. Set this file as the startup file by right clicking on it and choosing startup file. The following program shows how different matrix operations can be carried using Pytorch.

```
import sys
import torch

def main():
    a = torch.arange(2*2).reshape(2,2)
    print(a)
    b = torch.arange(2*2).reshape(2,2)
    print(b)
    c = a * b
    print(c) # element by element multiplication
    d = a + b # matrix addition
    print(d)
    e = torch.arange(2*3).reshape(2,3)
    print(e)
    f = torch.matmul(a,e) # matrix multiplication
    print(f)
    f1 = a @ e # also does matrix multiplication
    print(f1)
    g = torch.transpose(f,0,1)
    print(g)
    #f1 = f.reshape(1,f.shape[0], f.shape[1]) also works

    f1 = torch.unsqueeze(f,dim=0) # add a dimension in the beginning
    print(f1.shape)
    f1t = torch.transpose(f1,1,2)
    print(f1t)
    #-----batch matrix mult-----
    tensor1 = torch.randn(10, 3, 4)
    tensor2 = torch.randn(10, 4, 5)
    res = torch.matmul(tensor1, tensor2)
    print(res.shape)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```



Put a break point after each print statement and see the output by running the program in the debug mode i.e., from the menu choose, Debug->Start Debugging.

 A screenshot of a Python code editor showing the following code:

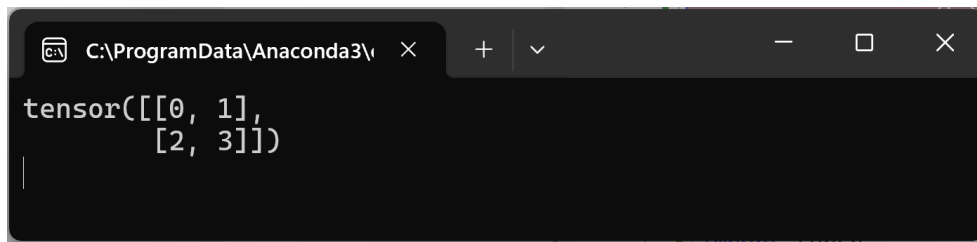

```

1 import sys
2 import torch
3
4 def main():
5     a = torch.arange(2*2).reshape(2,2)
6     print(a)
7     b = torch.arange(2*2).reshape(2,2)
8     print(b)
  
```

 A red circular break point is set on line 7. A blue arrow points from the text above to this break point.

Once the program stops at the break point, it will turn yellow, then you can examine the output by viewing the console.

A screenshot of the Visual Studio code editor showing the same Python code as before. The code is now highlighted in yellow, indicating it has stopped at the break point. A yellow lightbulb icon is visible next to line 7. The console window at the bottom is not visible in this screenshot.



```

C:\ProgramData\Anaconda3\... X + - □ X
tensor([[0, 1],
        [2, 3]])

```

For implementing more complex transformer architectures, we use the Einops library that provides easy tensor operations such as multiplying matrices, taking transposes, rearranging tensors in different dimensions. Add a python file called **EinopsTest.py** with the following code in it. Set this file as the startup file, and test the output of each operation by setting the breakpoint after each print statement.

The general philosophy of Einops is that if the input tensor is 2 dimensional, then you will choose two letters to describe the dimensions of the tensor e.g., *ij*. If you wanted to multiply element by element two 2-D tensors A, B, and add all the elements, the einops will appear as:

```
C = torch.einsum('ij, ij -> ', A, B)
```

If a variable does not appear on the output side i.e., right of \rightarrow , then a sum is carried out in the missing dimension(s).

```

import sys

import torch
from einops import rearrange

def main():
    A = torch.tensor([[1, 2, 3, 4],
                      [5, 6, 7, 8],
                      [9, 10, 11, 12],
                      [13, 14, 15, 16]])
    B = torch.tensor([[1, 2, 1, 1],
                      [3, 4, 2, 5],
                      [1, 3, 6, 7],
                      [1, 4, 6, 8]])
    print(A)
    print(B)

    C = torch.einsum('ij, jk -> ik', A, B) # matrix mult.
    print(C)

    C1 = torch.einsum('ij, jk -> ik', A, B) # matrix mult.
    print(C1)
    C2 = torch.einsum('ij, kj -> ik', A, B) # Ax(transpose(B) - matrix mult.
    print(C2)

    C3 = torch.einsum("ii -> i", A) # diagonal elements only
    print(C3)

```

```

C4 = torch.einsum("ii -> ", A) # sum diagonal elements - trace
print(C4)

C5 = torch.einsum("ij -> j", A) # sum column elements (row wise sum)
print(C5)

C6 = torch.einsum('ij, ij -> ij', A, B) # element wise product
print(C6)

C7 = torch.einsum('ij, ij, ij -> ij', A, A, A) # cube elements
print(C7)

C8 = torch.einsum('ij -> ji', A) # transpose
print(C8)

C9 = torch.einsum('ij,ij -> i', A, B) # multiply row wise and add each row
print(C9)

d1 = torch.tensor([3, 5, 7, 9])
d2 = torch.tensor([1, 2, 3, 4])
douter = torch.einsum('i, j -> ij', d1, d2) # outer product
print(douter)
dinner = torch.einsum('i, i -> ', d1, d2) # inner product
print(dinner)

dfrobenius = torch.einsum("ij, ij -> ", A, A) # frobenius norm
# sum of squares of all elements of a matrix
print('Frobenius norm...')
print(dfrobenius)

batch_tensor_1 = torch.arange(2 * 4 * 3).reshape(2, 4, 3)
print(batch_tensor_1)
batch_tensor_2 = torch.arange(2 * 4 * 3).reshape(2, 3, 4)
print(batch_tensor_2)
dmul = torch.einsum('bij, bjk -> bik', batch_tensor_1, batch_tensor_2) #
batch matrix multiplication
print(dmul)

dt = torch.randn((3,5,4,6,8,2,7,9)) # 8 dimensions
print(dt.shape)
esum = torch.einsum("ijklmnop -> p", dt)
# marginalize or sum over dim p
print(esum) # produces 9 numbers, try op instead of p

kv = torch.zeros((2,1024,64)) # 2 is batch size
q = torch.zeros((2,1024,64))
q2 = rearrange(q, 'b (n s) e->b n s e', s=16)
print(q2.shape) #[2,64,16,64]
q3 = rearrange(q2, 'b n s e-> (b n) s e')
print(q3.shape) #[128,16,64]

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

For implementing a transformer layer, we pass the input (assuming it has been tokenized and passed through an embedding layer with position encoding added) as a tensor of dimension batch, sequence length, embedding size e.g., 4, 100, 512. As we create a transformer, the operations needed are:

$$\text{Attention} = Q.K^T$$

$$\text{Output} = \text{Attention}.V$$

Further, the embedding dimension is divided into as many parts as the number of heads in each attention head i.e., if the embedding dimensionality is 512, and we decide to use 8 heads, then each head operates on size of 64. To show, how einops can be used to easily create a transformer layer, add another file to the project called TransformerLayer.py with the following code in it.

```
import sys
import torch
from torch import nn
from einops import rearrange

class TransformerLayer(nn.Module):
    def __init__(self, d) -> None:
        super().__init__()

        self.qkv = nn.Linear(d, d*3)
        self.wo = nn.Linear(d, d)

    def forward(self, x):
        x = self.qkv(x)
        q, k, v = tuple(rearrange(x, 'b n (k d h)->k b h n d', k=3, h=8))
        attn = torch.einsum('b h i k, b h j k->b h i j', q, k)
        out = torch.einsum('b h i k, b h k j->b h i j', attn, v)
        out = rearrange(out, 'b h n d->b n (h d)')
        out = self.wo(out)
        return out

def main():
    net = TransformerLayer(512)
    x = torch.rand((4, 100, 512))
    z = net(x)
    print(z.shape)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Examine the above code carefully, set the TransformerLayer.py as the startup file and step through the program to see how each step is transforming the data.

In the implementation of the transformer, we also use LayerNorm to normalize the data along the embedding dimension. To understand what exactly layer norm does, add a file called LayerNormTest.py with the following code in it.

```

import sys
import torch
import numpy as np

class NNLN(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.LN = torch.nn.LayerNorm(4)

    def forward(self, x):
        out = self.LN(x)
        return out

def main():
    x1 = np.arange(4)
    st = np.std(x1)
    mn = np.mean(x1)
    x2 = (x1 - mn)/st
    #print(x1, ' ', st)
    print('-----manual normalization')
    print(x2)
    d = torch.arange(4).float()
    print(d)
    x = d.view(1,-1)
    net = NNLN()
    z = net(x)
    print(z)

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

As you can see from the above code, the LayerNorm built into pytorch performs equivalent of subtracting the mean and dividing by the standard deviation along the embedding dimension. In reality, it has two other learnable parameters (similar to batch norm), but the fundamental concept is to normalize each embedding vector along the embedding dimension.

In implementing the masked attention (for generation purposes) so that future tokens are not attended to, we need to create a triangular mask such that the upper triangular part of the matrix is changed to $-\infty$. The following code demonstrates how that can be easily achieved. Add a file called TriuTest.py with the following code.

```

import sys
import torch
import numpy as np

def main():
    i = 4
    j = 4
    mask = torch.ones(i, j, device = 'cuda').triu_(1).bool() # try with
    triu_(0)
    print(mask)

```

```

print('\n')
attn = torch.rand((4,4)).cuda()
print(attn)
print('\n')
attn_masked = attn.masked_fill(mask,-np.inf)
print(attn_masked)

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

In implementing the generation part of the transformer, we take the top k choices (highest top k values after applying softmax to the logits layer).

Add a file called MultinomialTest.py with the following code in it.

```

import sys
import torch
import torch.nn.functional as F

def main():
    logits = torch.tensor([1, 2, 3, 1, 3, 2, 3], dtype=torch.float)
    s = F.softmax(logits, dim=0)
    # to simulate top-k, lets zero out a few entries.
    s[1] = 0
    s[6] = 0
    s[1] = 0
    print(s)
    index1 = torch.multinomial(s, 1) # return probabilistically index of one top
choice
    print(index1)
    index2 = torch.multinomial(s, 2) # index of top 2 choices
    print(index2)

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

If you run the program after setting the MultinomialTest.py as the startup file, you will see that the index of the highest probable value is often returned by the multinomial function call.

```

C:\WINDOWS\system32\cmd. x + v
tensor([0.0338, 0.0000, 0.2496, 0.0338, 0.2496, 0.0918, 0.0000])
tensor([2])
tensor([2, 4])
Press any key to continue . . . |

```

For transformer implementation, often we use the cross entropy loss which is defined as:

$$CE = - \sum_{i=1}^{i=N} y_i \cdot \log(\hat{y}_i)$$

Where N is the number of classes and \hat{y} is the actual output and y is the target.

Pytorch also defines negative log likelihood loss which is similar to cross entropy. To get a better understanding of these, add a file called CrossEntropyTest.py with the following code in it. The comments in the program clarify the loss calculations.

```
import sys
import numpy as np
import torch

def main():
    a = torch.arange(10)
    a2 = a.view(-1,10)
    print(a2)
    a3 = a2[:, -3:]
    print(a3)

    # assume 3 outputs and batch size of 2, so logits = 2x3 tensor
    logits=torch.tensor([[1,3.0,5],[2,4.0,1]])
    # above indicates predicted output is 2 and 1 (index of highest value)
    print('-----logits-----')
    print(logits)

    targets=torch.tensor([2,0])
    # targets are specified as long, i.e.,
    # index of which output is to be recognized, try with [2,1] to see if loss
    # decreases
    # pytorch's cross entropy loss, operates on logits (not on softmax layer)
    loss = torch.nn.functional.cross_entropy(logits,targets)
    print('\ncross entropy loss by pytorch=', loss)

    # pytorch's nll_loss (negative log likelihood loss) is similar to cross
    # entropy
    # it operates on log_softmax, rather than raw logits
    outs = torch.softmax(logits,dim=1)
    print('-----softmax-----')
    print(outs)
    outs2 = torch.nn.functional.log_softmax(logits, dim=1)
    loss_nll = torch.nn.functional.nll_loss(outs2,targets)
    print('nll loss by pytorch =',loss_nll)

    # compute cross entropy ourselves
    z = (np.log(outs[0,targets[0]]) + np.log(outs[1,targets[1]]))/2
    print("\ncross entropy by our calculation=", -z)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```


Sometimes, we use the KL divergence (Kullback-Leibler divergence) which measures the similarity

$$D_{\text{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

To better understand KL divergence, add a file to the project called `KLDivergenceTest.py` with the following code in it.

```
import numpy as np
import sys

def kl(p, q):
    #Kullback-Leibler divergence D(P || Q) for discrete distributions
    return np.sum(np.where(q != 0, p * np.log(p / q), 0))

def main():
    p = np.array([0.8, 0.1, 0.05, 0.05]) # for a distribution, sum should be 1
    q = np.array([0.2, 0.3, 0.3, 0.2]) # 0.84

    # the following two distributions are closer to each other
    # so KL divergence will be smaller, uncomment following to test it
    # p = np.array([0.8, 0.1, 0.05, 0.05])
    # q = np.array([0.85, 0.05, 0.05, 0.05]) # 0.0208
    res = kl(p,q)
    print('KL divergence =', res)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

If you set the above file as startup file and run the program, it will show the KL divergence between the two distributions as:



The screenshot shows a Windows command prompt window with the title bar 'C:\WINDOWS\system32\cmd.' and standard window controls. The command prompt displays the output of the program: 'KL divergence = 0.8402715685117041' followed by 'Press any key to continue . . . |'.

If you uncomment the last section, i.e.,

```
import numpy as np
import sys

def kl(p, q):
    #Kullback-Leibler divergence D(P || Q) for discrete distributions
    return np.sum(np.where(q != 0, p * np.log(p / q), 0))

def main():
    # p = np.array([0.8, 0.1, 0.05, 0.05]) # for a distribution, sum should be 1
```

```
# q = np.array([0.2, 0.3, 0.3, 0.2]) # 0.84

# the following two distributions are closer to each other
# so KL divergence will be smaller, uncomment following to test it
p = np.array([0.8, 0.1, 0.05, 0.05])
q = np.array([0.85, 0.05, 0.05, 0.05]) # 0.0208
res = kl(p,q)
print('KL divergence =', res)

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Now the KL divergence will have a lower value.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.' and standard window controls. The command prompt displays the output 'KL divergence = 0.020815020602846734' followed by a prompt 'Press any key to continue . . . |'.