Deep Learning

Cross Entropy Loss in PyTorch

Question: for Cross Entropy Loss calculation in PyTorch, why the input doesn't need to be probabilities when the CE calculation requires probabilities?

Answer: the PyTorch calculation is a little bit different, it has SoftMax calculation built in in the cross-entropy loss calculation

Cross entropy loss (summation is over all classes)

$$H(p,q) = -\sum p(x) * \log\big(q(x)\big)$$

Where $q(x)$ is the predicted probability after SoftMax calculation

How PyTorch calculates cross entropy loss (summations are overall classes)

$$l_n = -\sum w_c * \log\frac{e^{x_{n,c}}}{\sum e^{x_{n,i}}} * y_{n,c}$$

Precision and Recall

Question: what is the difference between precision and recall and accuracy?

Answer:

- Accuracy is the number of correct classifications out of all samples
- Recall is the number of correct positive cases identified out of all positive cases

$$recall = \frac{true\ positives}{true\ positives\ + false\ negatives} = \frac{terrorists\ correctly\ identified}{terrorists\ correctly\ identified + terrorists\ incorrectly\ labeled\ as\ not\ terrorists}$$

- Precision is the number of correct positive cases identified out of positive cases identified by the model

$$precision = \frac{true\ positives}{true\ positives + false\ positives} = \frac{terrorists\ correctly\ identified}{terrorists\ correctly\ identified + individuals\ incorrectly\ labeled\ as\ terroists}$$

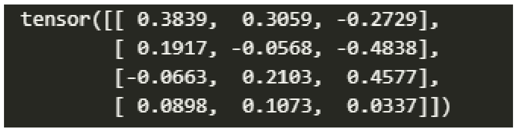
Torch.nn.Embedding

Syntax: $torch.nn.embedding(vocab\_size, embed\_dim)$

- Each value in the input_text must be < vocabulary size
- The layer takes each value and convert the value from one number to vector embedding of size embed_dim
- The layer is basically a lookup table, the value represent the index to retrieve the vector embedding

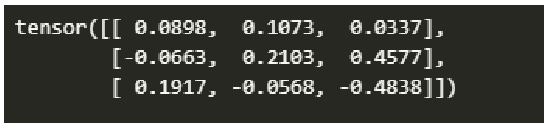
Assuming vocabulary size of 4 and embedding dimension of 3

- Given the weight matrix



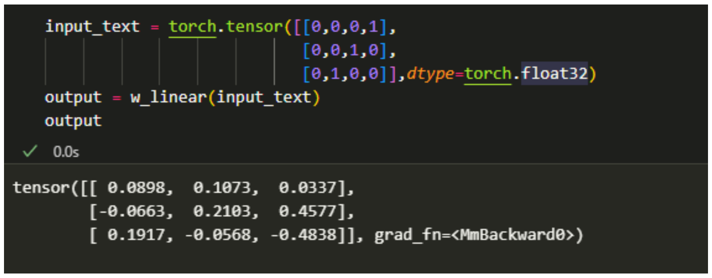
```
tensor([[ 0.3839,  0.3059, -0.2729],
        [ 0.1917, -0.0568, -0.4838],
        [-0.0663,  0.2103,  0.4577],
        [ 0.0898,  0.1073,  0.0337]])
```

- The input text of [3,2,1] will be embedded into



```
tensor([[ 0.0898,  0.1073,  0.0337],
        [-0.0663,  0.2103,  0.4577],
        [ 0.1917, -0.0568, -0.4838]])
```

- This is equivalent to one hot encode the input and feed it to a linear layer with the transpose of the given weight matrix



```
input_text = torch.tensor([[0,0,0,1],
                           [0,0,1,0],
                           [0,1,0,0]],dtype=torch.float32)
output = w_linear(input_text)
output
```

✓ 0.0s

```
tensor([[ 0.0898,  0.1073,  0.0337],
        [-0.0663,  0.2103,  0.4577],
        [ 0.1917, -0.0568, -0.4838]], grad_fn=<MmBackward0>)
```

Param Initialization Part 1: Expectation of Initial Loss for classification

- Assume n class, and parameters are initialized to predict roughly equal probability for each class, the cross entropy loss would be

$$-\ln(1/n)$$

Param Initialization Part 2: Effect of highly skewed initial prediction

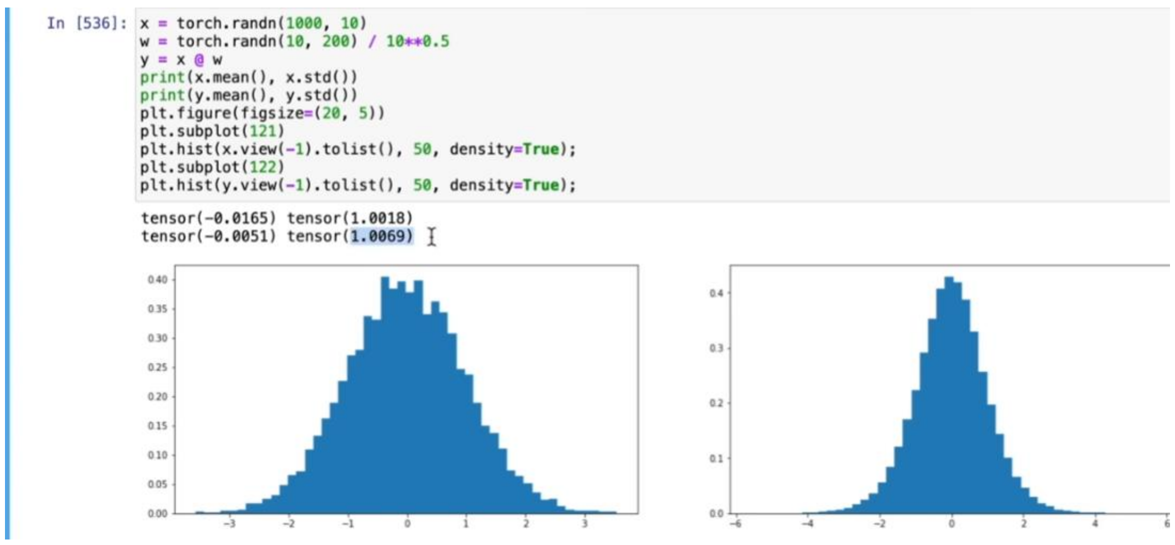
- When initial model output is highly skewed (not evenly distributed across all classes), the model will take a lot of unnecessary training to first make the distribution even, then train toward the correct class.
    - The outcome is so that the output logit (model output before the softmax layer or sigmoid layer) is small (closer to zero)
    - Hence, multiple the initial bias by 0 (initialize all bias to 0), multiple the initial W by 0.1 or 0.01
    - Why not set W to 0? It can mask an incorrect implementation of a gradient

Param Initialization Part 3: consequence of highly activate tanh activation

- When tanh is high activated, that implies that there are many extreme values got squashed into negative one and one, in this case, those values are highly inelastic – tweaking the parameters would not change the outcome after the activation, those neurons are close to be "dead"
    - This is like a permanent brain damage: the neuron will never activate because it is in the extreme of the manifold
    - In this case, we want to make sure that the distribution of the data feed into the activation (at least initially) is small. This means the weights should be very small and close to 0

Param Initialization Part 4: how to initialize

- How to preserve the input standard deviation with matrix multiplication (what number should we multiple W with?)



```
In [536]: x = torch.randn(1000, 10)
          w = torch.randn(10, 200) / 10**0.5
          y = x @ w
          print(x.mean(), x.std())
          print(y.mean(), y.std())
          plt.figure(figsize=(20, 5))
          plt.subplot(121)
          plt.hist(x.view(-1).tolist(), 50, density=True);
          plt.subplot(122)
          plt.hist(y.view(-1).tolist(), 50, density=True);

          tensor(-0.0165) tensor(1.0018)
          tensor(-0.0051) tensor(1.0069)
```

- $1/\sqrt{n}$ – n is the number of features / inputs

- Paper:
  - Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015)
- PyTorch:
  - Kaiming_normal

Batch Normalization

- It is unnatural because the model is supposed to process a single input, we train them in batches because of optimization. Batch normalization normalizes the layer output for each batch, then use parameters to make shifts and scale to the distribution.
  - The layer output of a sample will be a little different when it is trained in a different batch
  - This acts like a regularization (a little like data augmentation)

- For inference time
  - Idea 1: calibrate the batch norm at the end of training - Calculate the mean and std over the entire training set to be used at inference time
  - **Idea 2**: keep track of the running stats – initialize running mean at 0 and running std at 1 with size equals to (1, hidden dim)

$$with\ torch.no\_grad$$

$$mean_{running} = 0.999 \times mean_{running} + 0.001 \times mean_{current\ batch}$$

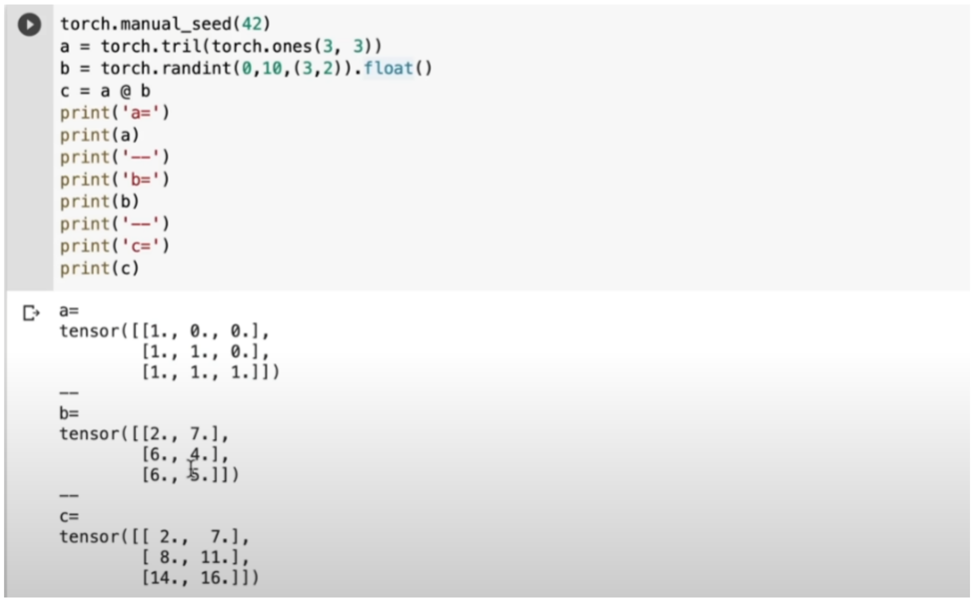
    - Std calculation is the same
    - Receives a small update every time

        - The 0.001 is the momentum, roughly speaking, if you have a large batch size, we can use higher momentum, because the batch mean and std will be roughly the same for different batches if the batch size is large

- The layer before the batch norm should not have bias term
    - Since we are standardizing the data with batch normalization, adding the bias term is a waste, for example, the bias shifts the distribution rightward, then the normalization step will shift it leftward if the distribution is centered on a positive value

Transformer Part 1: accumulate information from previous tensors

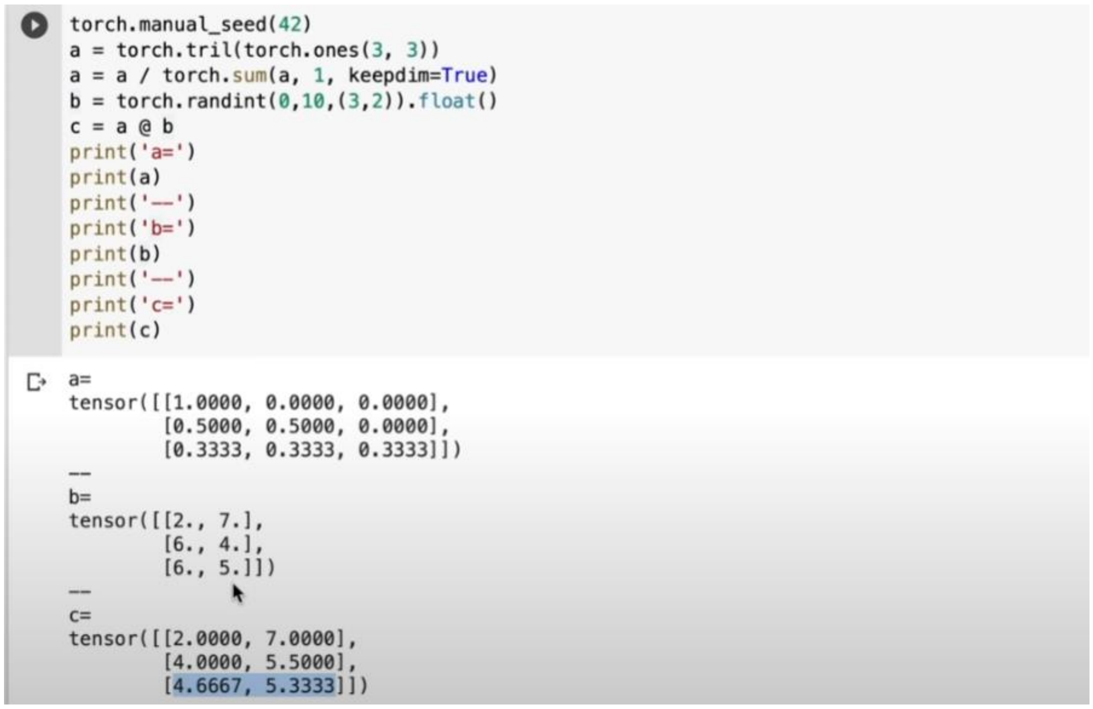
- Get sum over previous tensors (element-wise) up until current tensor



```
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3))
b = torch.randint(0,10,(3,2)).float()
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
```

```
a=
tensor([[1., 0., 0.],
        [1., 1., 0.],
        [1., 1., 1.]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[ 2.,  7.],
        [ 8., 11.],
        [14., 16.]])
```

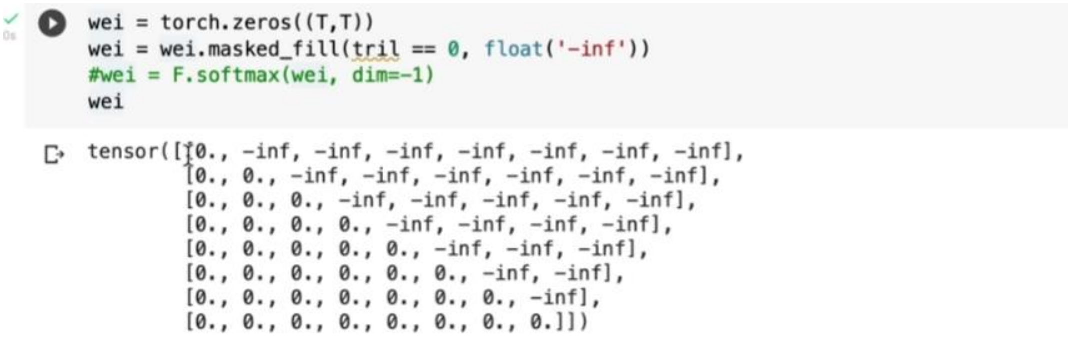
- Average



```
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3))
a = a / torch.sum(a, 1, keepdim=True)
b = torch.randint(0,10,(3,2)).float()
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
```

```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

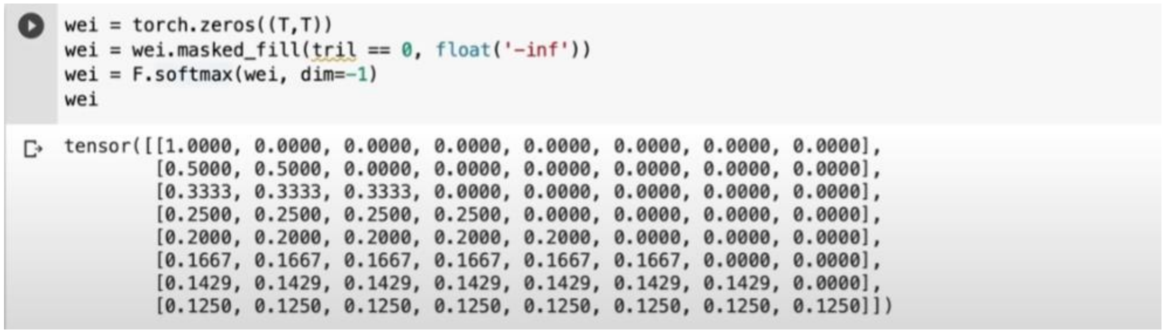
- More sophisticated method



```
wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
#wei = F.softmax(wei, dim=-1)
wei
```

```
tensor([[0., -inf, -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., -inf, -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., -inf, -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., -inf, -inf, -inf, -inf],
        [0., 0., 0., 0., 0., -inf, -inf, -inf],
        [0., 0., 0., 0., 0., 0., -inf, -inf],
        [0., 0., 0., 0., 0., 0., 0., -inf],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

  - With softmax



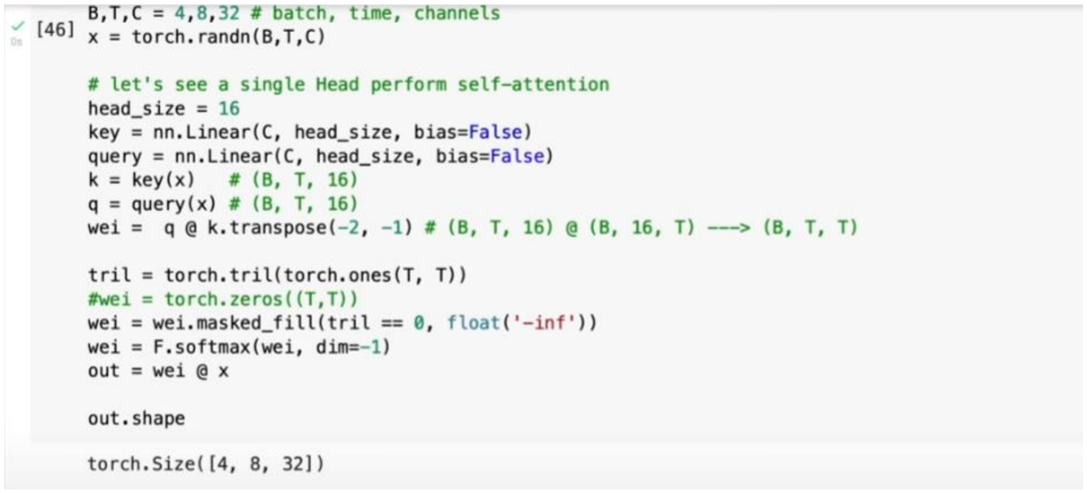
```
wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)
wei
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.3333, 0.3333, 0.3333, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2000, 0.2000, 0.2000, 0.2000, 0.2000, 0.0000, 0.0000, 0.0000],
        [0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.0000, 0.0000],
        [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.0000],
        [0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250, 0.1250]])
```

Transformer Part 2: self-attention – key and query

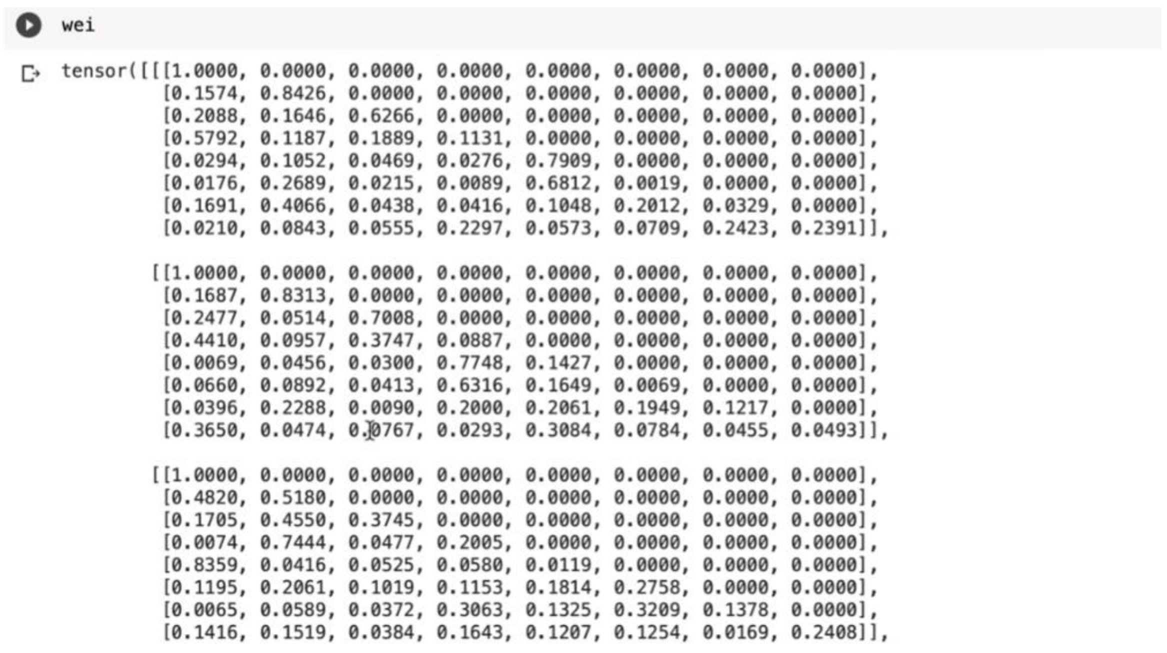
- Instead of using average, we want to accumulate the past information in a data-dependent way – self-attention;
- Every single token at each position will emit two vectors
    - Query: represents what am I looking for?
    - Key: represents what do I contain?
- Perform dot product between keys and queries
    - Use the query of current token, dot product of all the keys of other tokens
    - The result of the dot product becomes the weight matrix (*wei* and *a* in part 1)
    - If a key is highly aligned with my current query, they will interact with a high amount, and I get to learn more about that specific token as opposed to every other token in the sequence

- Implementation – mask information as the second step



```
[46]
B,T,C = 4,8,32 # batch, time, channels
x = torch.randn(B,T,C)

# let's see a single Head perform self-attention
head_size = 16
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
k = key(x)   # (B, T, 16)
q = query(x) # (B, T, 16)
wei =  q @ k.transpose(-2, -1) # (B, T, 16) @ (B, 16, T) ---> (B, T, T)

tril = torch.tril(torch.ones(T, T))
#wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)
out = wei @ x

out.shape

torch.Size([4, 8, 32])
```

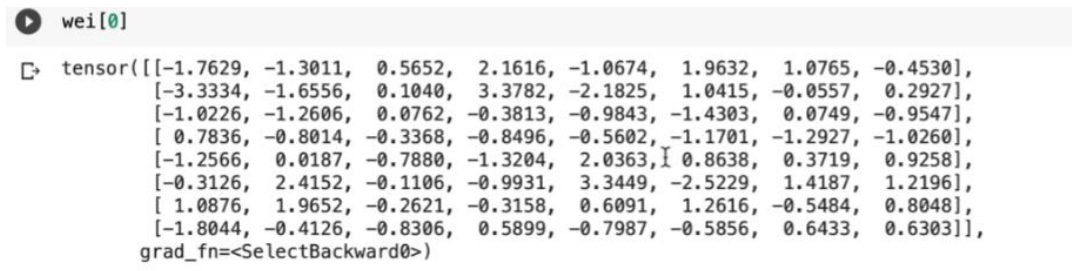
    - Now the weights are data-dependent, the weights are different for each batch because there are different tokens in the each batch



```
wei
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
         [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
         [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
         [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],

        [[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1687, 0.8313, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.2477, 0.0514, 0.7008, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.4410, 0.0957, 0.3747, 0.0887, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0069, 0.0456, 0.0300, 0.7748, 0.1427, 0.0000, 0.0000, 0.0000],
         [0.0660, 0.0892, 0.0413, 0.6316, 0.1649, 0.0069, 0.0000, 0.0000],
         [0.0396, 0.2288, 0.0090, 0.2000, 0.2061, 0.1949, 0.1217, 0.0000],
         [0.3650, 0.0474, 0.0767, 0.0293, 0.3084, 0.0784, 0.0455, 0.0493]],

        [[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.4820, 0.5180, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.1705, 0.4550, 0.3745, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.0074, 0.7444, 0.0477, 0.2005, 0.0000, 0.0000, 0.0000, 0.0000],
         [0.8359, 0.0416, 0.0525, 0.0580, 0.0119, 0.0000, 0.0000, 0.0000],
         [0.1195, 0.2061, 0.1019, 0.1153, 0.1814, 0.2758, 0.0000, 0.0000],
         [0.0065, 0.0589, 0.0372, 0.3063, 0.1325, 0.3209, 0.1378, 0.0000],
         [0.1416, 0.1519, 0.0384, 0.1643, 0.1207, 0.1254, 0.0169, 0.2408]],
```

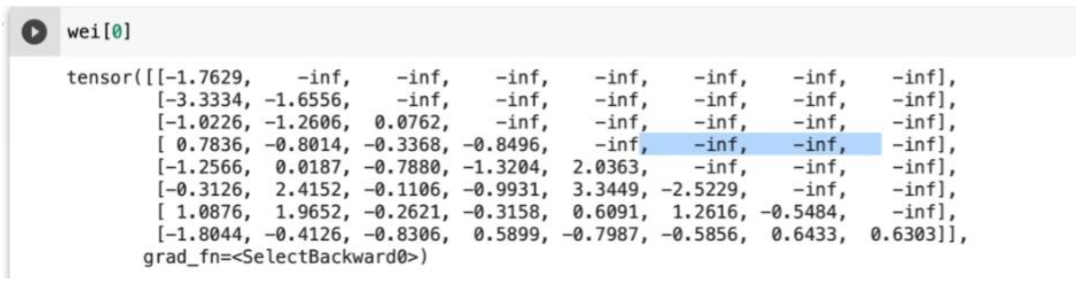
- Stages of weight matrix
    - Without masking the weights



```
wei[0]
```

```
tensor([[-1.7629, -1.3011,  0.5652,  2.1616, -1.0674,  1.9632,  1.0765, -0.4530],
        [-3.3334, -1.6556,  0.1040,  3.3782, -2.1825,  1.0415, -0.0557,  0.2927],
        [-1.0226, -1.2606,  0.0762, -0.3813, -0.9843, -1.4303,  0.0749, -0.9547],
        [ 0.7836, -0.8014, -0.3368, -0.8496, -0.5602, -1.1701, -1.2927, -1.0260],
        [-1.2566,  0.0187, -0.7880, -1.3204,  2.0363,  0.8638,  0.3719,  0.9258],
        [-0.3126,  2.4152, -0.1106, -0.9931,  3.3449, -2.5229,  1.4187,  1.2196],
        [ 1.0876,  1.9652, -0.2621, -0.3158,  0.6091,  1.2616, -0.5484,  0.8048],
        [-1.8044, -0.4126, -0.8306,  0.5899, -0.7987, -0.5856,  0.6433,  0.6303]],
       grad_fn=<SelectBackward0>)
```

    - After masking before softmax

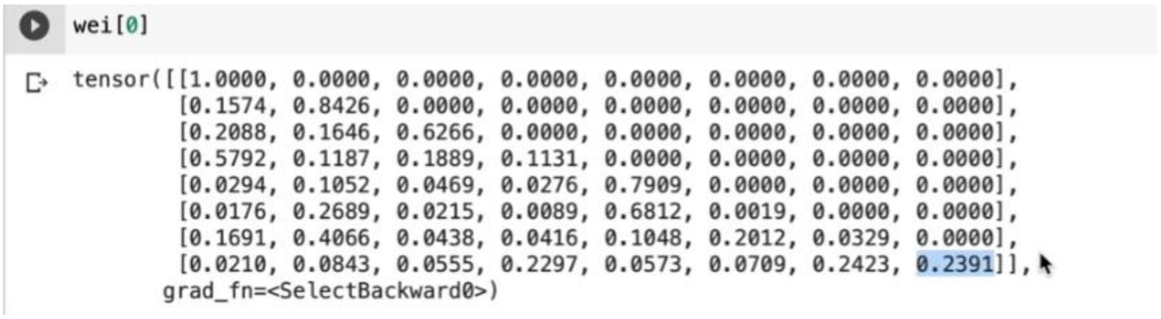


```
wei[0]
```

```
tensor([[-1.7629,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [-3.3334, -1.6556,    -inf,    -inf,    -inf,    -inf,    -inf,    -inf],
        [-1.0226, -1.2606,  0.0762,    -inf,    -inf,    -inf,    -inf,    -inf],
        [ 0.7836, -0.8014, -0.3368, -0.8496,    -inf,    -inf,    -inf,    -inf],
        [-1.2566,  0.0187, -0.7880, -1.3204,  2.0363,    -inf,    -inf,    -inf],
        [-0.3126,  2.4152, -0.1106, -0.9931,  3.3449, -2.5229,    -inf,    -inf],
        [ 1.0876,  1.9652, -0.2621, -0.3158,  0.6091,  1.2616, -0.5484,    -inf],
        [-1.8044, -0.4126, -0.8306,  0.5899, -0.7987, -0.5856,  0.6433,  0.6303]],
       grad_fn=<SelectBackward0>)
```

Transformer Part 3: Examining self-attention weights

- Now let's look at the attention weights of the first batch

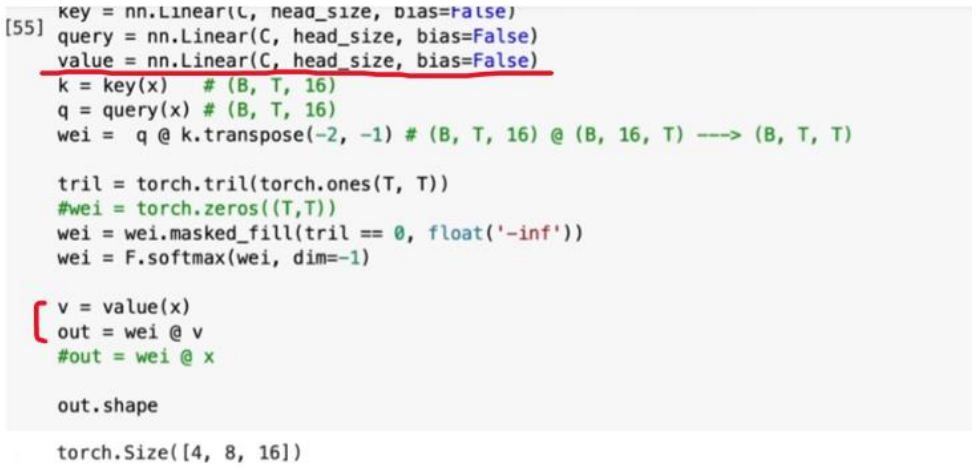


```
wei[0]

tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
        [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
        [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
        [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
       grad_fn=<SelectBackward0>)
```

  - The first tensor represents the "importance" of other tokens in the batch to the first token, and since there's only future tokens, the only significant tokens to the first token is itself, therefore, there's only a 1 at the first position
  - The second tensor represents the "importance" of other tokens in the batch to the second token, and we can see that there's some importance of the first token to the second token
  - The last tensor – see it as the eighth token emitted query to all the other tokens before, all the other tokens provide keys to the query, then based on the dot product and then softmax, we can see how important each previous token is to the eighth token

Transformer Part 4: self-attention – value

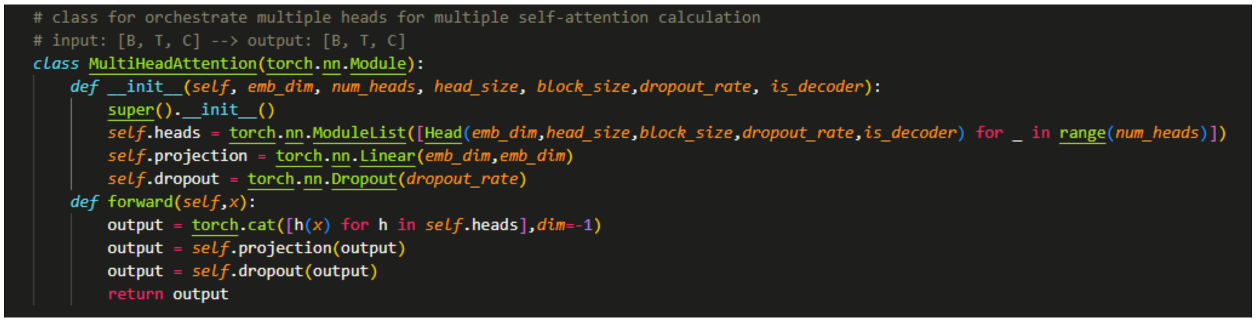
- We don't apply the weight matrix on the input directly (the input is regarded as a private information to the token)



```
[55] key = nn.Linear(C, head_size, bias=False)
     query = nn.Linear(C, head_size, bias=False)
     value = nn.Linear(C, head_size, bias=False)
     k = key(x)   # (B, T, 16)
     q = query(x) # (B, T, 16)
     wei =  q @ k.transpose(-2, -1) # (B, T, 16) @ (B, 16, T) ---> (B, T, T)

     tril = torch.tril(torch.ones(T, T))
     #wei = torch.zeros((T,T))
     wei = wei.masked_fill(tril == 0, float('-inf'))
     wei = F.softmax(wei, dim=-1)

     v = value(x)
     out = wei @ v
     #out = wei @ x

     out.shape

     torch.Size([4, 8, 16])
```

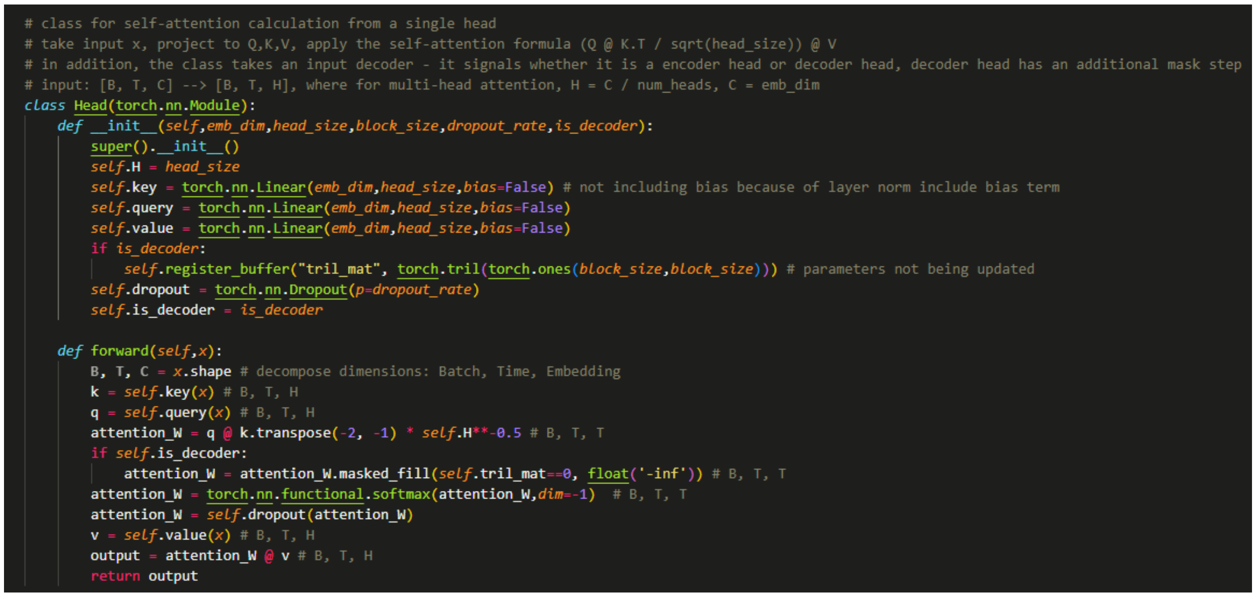
Transformer Part 5: multi-head attention & Implementation so far

- Multi-head attention



```python
# class for orchestrate multiple heads for multiple self-attention calculation
# input: [B, T, C] --> output: [B, T, C]
class MultiHeadAttention(torch.nn.Module):
    def __init__(self, emb_dim, num_heads, head_size, block_size,dropout_rate, is_decoder):
        super().__init__()
        self.heads = torch.nn.ModuleList([Head(emb_dim,head_size,block_size,dropout_rate,is_decoder) for _ in range(num_heads)])
        self.projection = torch.nn.Linear(emb_dim,emb_dim)
        self.dropout = torch.nn.Dropout(dropout_rate)
    def forward(self,x):
        output = torch.cat([h(x) for h in self.heads],dim=-1)
        output = self.projection(output)
        output = self.dropout(output)
        return output
```

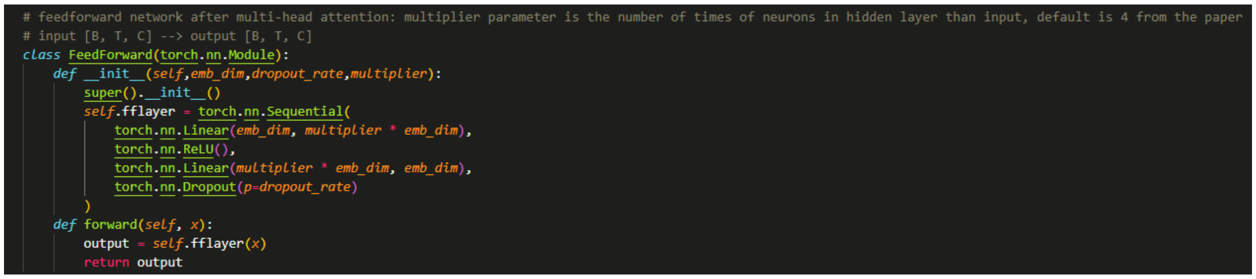
- Where Head is defined as



```python
# class for self-attention calculation from a single head
# take input x, project to Q,K,V, apply the self-attention formula (Q @ K.T / sqrt(head_size)) @ V
# in addition, the class takes an input decoder - it signals whether it is a encoder head or decoder head, decoder head has an additional mask step
# input: [B, T, C] --> [B, T, H], where for multi-head attention, H = C / num_heads, C = emb_dim
class Head(torch.nn.Module):
    def __init__(self,emb_dim,head_size,block_size,dropout_rate,is_decoder):
        super().__init__()
        self.H = head_size
        self.key = torch.nn.Linear(emb_dim,head_size,bias=False) # not including bias because of layer norm include bias term
        self.query = torch.nn.Linear(emb_dim,head_size,bias=False)
        self.value = torch.nn.Linear(emb_dim,head_size,bias=False)
        if is_decoder:
            self.register_buffer("tril_mat", torch.tril(torch.ones(block_size,block_size))) # parameters not being updated
        self.dropout = torch.nn.Dropout(p=dropout_rate)
        self.is_decoder = is_decoder

    def forward(self,x):
        B, T, C = x.shape # decompose dimensions: Batch, Time, Embedding
        k = self.key(x) # B, T, H
        q = self.query(x) # B, T, H
        attention_W = q @ k.transpose(-2, -1) * self.H**-0.5 # B, T, T
        if self.is_decoder:
            attention_W = attention_W.masked_fill(self.tril_mat==0, float('-inf')) # B, T, T
        attention_W = torch.nn.functional.softmax(attention_W,dim=-1)  # B, T, T
        attention_W = self.dropout(attention_W)
        v = self.value(x) # B, T, H
        output = attention_W @ v # B, T, H
        return output
```

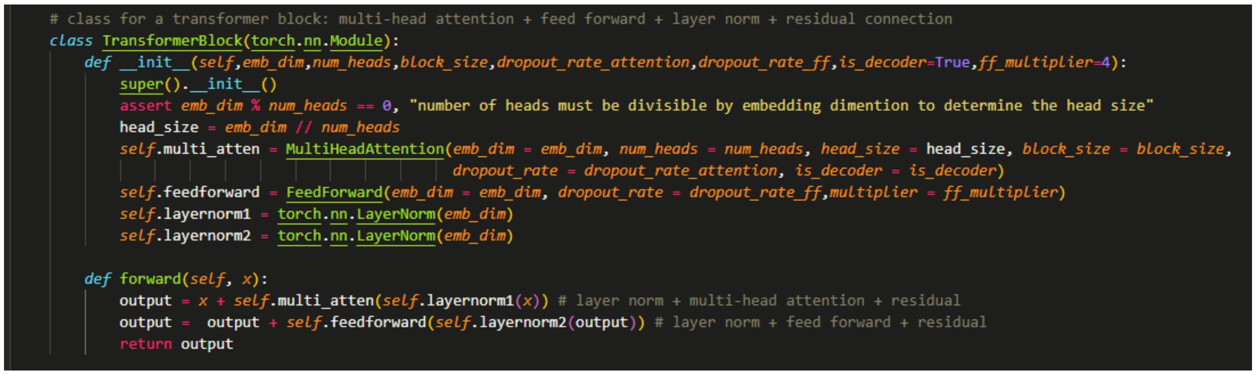
Transformer Part 6: Adding Feedforward layers

- Feedforward layer



```python
# feedforward network after multi-head attention: multiplier parameter is the number of times of neurons in hidden layer than input, default is 4 from the paper
# input [B, T, C] --> output [B, T, C]
class FeedForward(torch.nn.Module):
    def __init__(self,emb_dim,dropout_rate,multiplier):
        super().__init__()
        self.fflayer = torch.nn.Sequential(
            torch.nn.Linear(emb_dim, multiplier * emb_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(multiplier * emb_dim, emb_dim),
            torch.nn.Dropout(p=dropout_rate)
        )
    def forward(self, x):
        output = self.fflayer(x)
        return output
```

Transformer Part 7: Residual Connection, blocks, and layer normalization (apply before)



```python
# class for a transformer block: multi-head attention + feed forward + layer norm + residual connection
class TransformerBlock(torch.nn.Module):
    def __init__(self,emb_dim,num_heads,block_size,dropout_rate_attention,dropout_rate_ff,is_decoder=True,ff_multiplier=4):
        super().__init__()
        assert emb_dim % num_heads == 0, "number of heads must be divisible by embedding dimention to determine the head size"
        head_size = emb_dim // num_heads
        self.multi_atten = MultiHeadAttention(emb_dim = emb_dim, num_heads = num_heads, head_size = head_size, block_size = block_size,
                                              dropout_rate = dropout_rate_attention, is_decoder = is_decoder)
        self.feedforward = FeedForward(emb_dim = emb_dim, dropout_rate = dropout_rate_ff,multiplier = ff_multiplier)
        self.layernorm1 = torch.nn.LayerNorm(emb_dim)
        self.layernorm2 = torch.nn.LayerNorm(emb_dim)

    def forward(self, x):
        output = x + self.multi_atten(self.layernorm1(x)) # layer norm + multi-head attention + residual
        output =  output + self.feedforward(self.layernorm2(output)) # layer norm + feed forward + residual
        return output
```

Transformer Part 8: Positional Encoding

- Formula:
    - k = 0, …, block size – 1 [block size is sequence length]
    - i = 0, …, (embedding dimension /2) – 1

$$PE_{(k,2i)} = \sin\left(\frac{k}{10000^{\frac{2i}{emb\_dim}}}\right)$$

$$PE_{(k,2i+1)} = \cos\left(\frac{k}{10000^{\frac{2i}{emb\_dim}}}\right)$$

- example

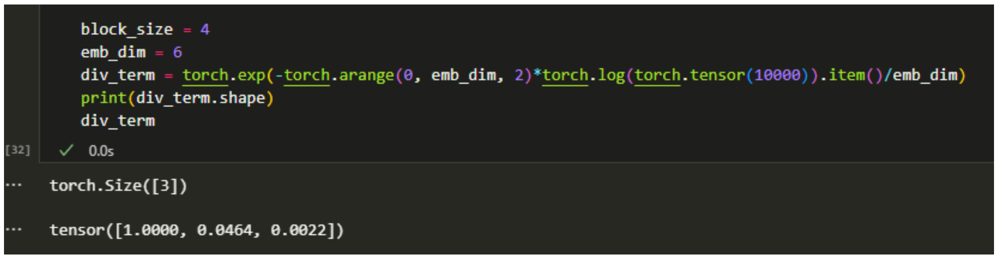
$$P(0) = \left[sin(\frac{0}{n^{\frac{0}{4}}}), cos(\frac{0}{n^{\frac{0}{4}}}), sin(\frac{0}{n^{\frac{2}{4}}}), cos(\frac{0}{n^{\frac{2}{4}}})\right]$$

$$P(1) = \left[sin(\frac{1}{n^{\frac{0}{4}}}), cos(\frac{1}{n^{\frac{0}{4}}}), sin(\frac{1}{n^{\frac{2}{4}}}), cos(\frac{1}{n^{\frac{2}{4}}})\right]$$

$$P(2) = \left[sin(\frac{2}{n^{\frac{0}{4}}}), cos(\frac{2}{n^{\frac{0}{4}}}), sin(\frac{2}{n^{\frac{2}{4}}}), cos(\frac{2}{n^{\frac{2}{4}}})\right]$$

- intuition:
  - For each token (i.e., k), the positional encoding generate unique sine and cosine waves that signifies the unique position the token is in
- implementation
  - Step 1: calculate $\frac{1}{10000^{\frac{2i}{emb\_dim}}}$ for all i, i.e., result length is $torch.size\left(\left[\frac{emb_{dim}}{2}\right]\right)$ with a trick

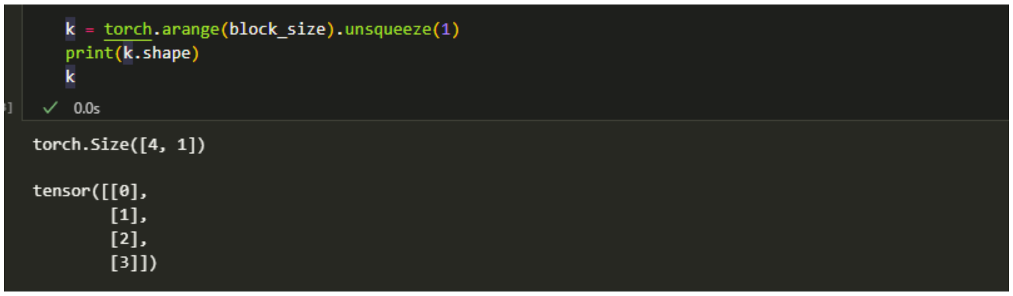
$$\frac{1}{n^{\frac{2i}{d_{model}}}} = n^{-\frac{2i}{d_{model}}} = e^{log(n^{-\frac{2i}{d_{model}}})} = e^{-\frac{2i}{d_{model}}log(n)} = e^{-\frac{2i\ log(n)}{d_{model}}}$$



```
block_size = 4
emb_dim = 6
div_term = torch.exp(-torch.arange(0, emb_dim, 2)*torch.log(torch.tensor(10000)).item()/emb_dim)
print(div_term.shape)
div_term
```

```
torch.Size([3])

tensor([1.0000, 0.0464, 0.0022])
```

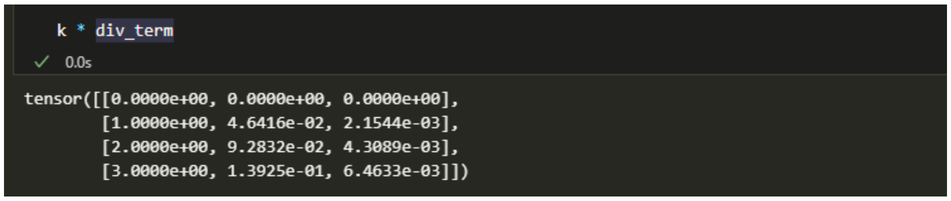
  - Step 2: construct the vector for k – it has to be a column vector whereas the previous calculation is row vector with shape $torch.size([block\_size, 1])$



```
k = torch.arange(block_size).unsqueeze(1)
print(k.shape)
k
```

```
torch.Size([4, 1])

tensor([[0],
        [1],
        [2],
        [3]])
```

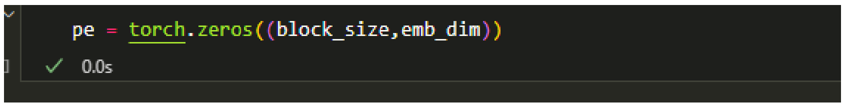
  - Step 3: produce the product between sine and cosine



```
k * div_term
```

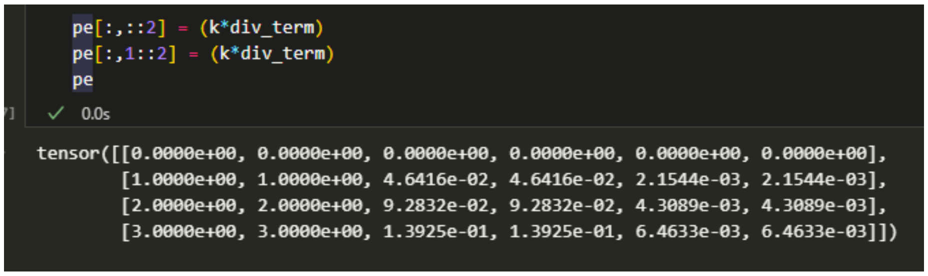
```
tensor([[0.0000e+00, 0.0000e+00, 0.0000e+00],
        [1.0000e+00, 4.6416e-02, 2.1544e-03],
        [2.0000e+00, 9.2832e-02, 4.3089e-03],
        [3.0000e+00, 1.3925e-01, 6.4633e-03]])
```

- Step 4: initialize positional encoding matrix with size $torch.size([block\_size, emb\_dim])$



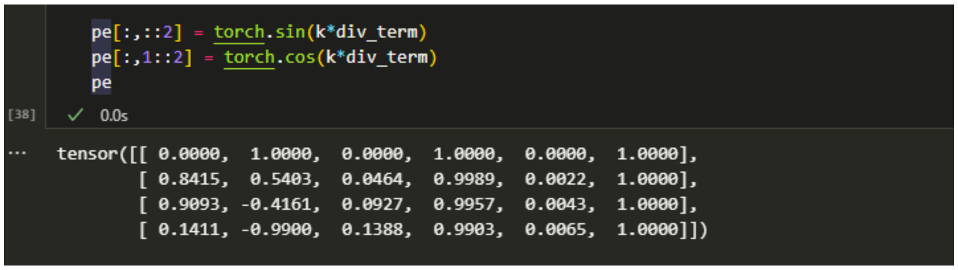
```
pe = torch.zeros((block_size,emb_dim))
✓ 0.0s
```

- Step 5: fill in without sine and cosine



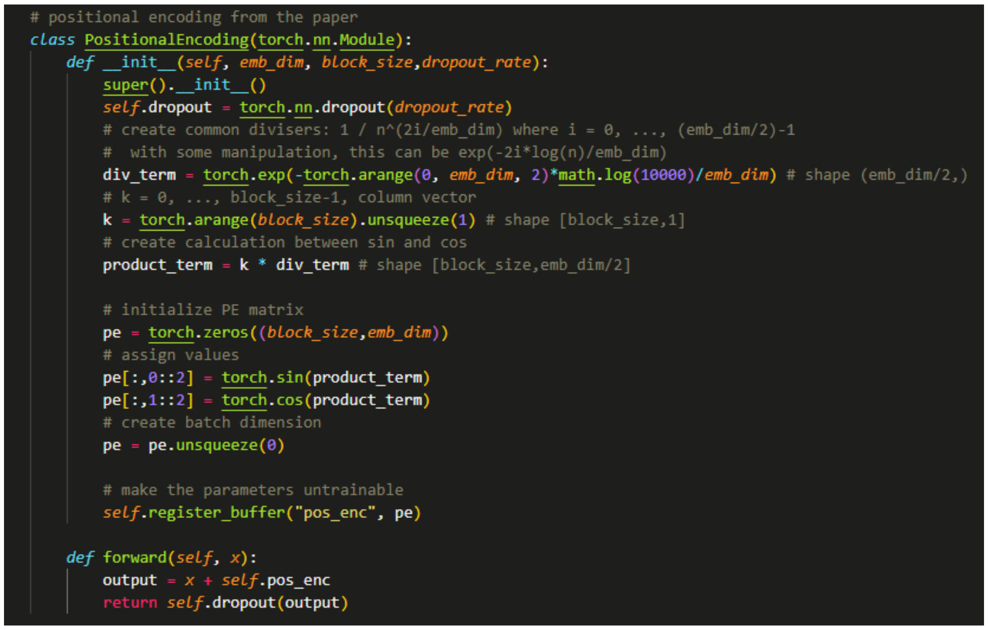
```
pe[:,::2] = (k*div_term)
pe[:,1::2] = (k*div_term)
pe
✓ 0.0s

tensor([[0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
        [1.0000e+00, 1.0000e+00, 4.6416e-02, 4.6416e-02, 2.1544e-03, 2.1544e-03],
        [2.0000e+00, 2.0000e+00, 9.2832e-02, 9.2832e-02, 4.3089e-03, 4.3089e-03],
        [3.0000e+00, 3.0000e+00, 1.3925e-01, 1.3925e-01, 6.4633e-03, 6.4633e-03]])
```

  - Notice pairs are the same, because the calculation is the same, only distinguished by sine and cosine



```
pe[:,::2] = torch.sin(k*div_term)
pe[:,1::2] = torch.cos(k*div_term)
pe
[38] ✓ 0.0s

tensor([[ 0.0000,  1.0000,  0.0000,  1.0000,  0.0000,  1.0000],
        [ 0.8415,  0.5403,  0.0464,  0.9989,  0.0022,  1.0000],
        [ 0.9093, -0.4161,  0.0927,  0.9957,  0.0043,  1.0000],
        [ 0.1411, -0.9900,  0.1388,  0.9903,  0.0065,  1.0000]])
```

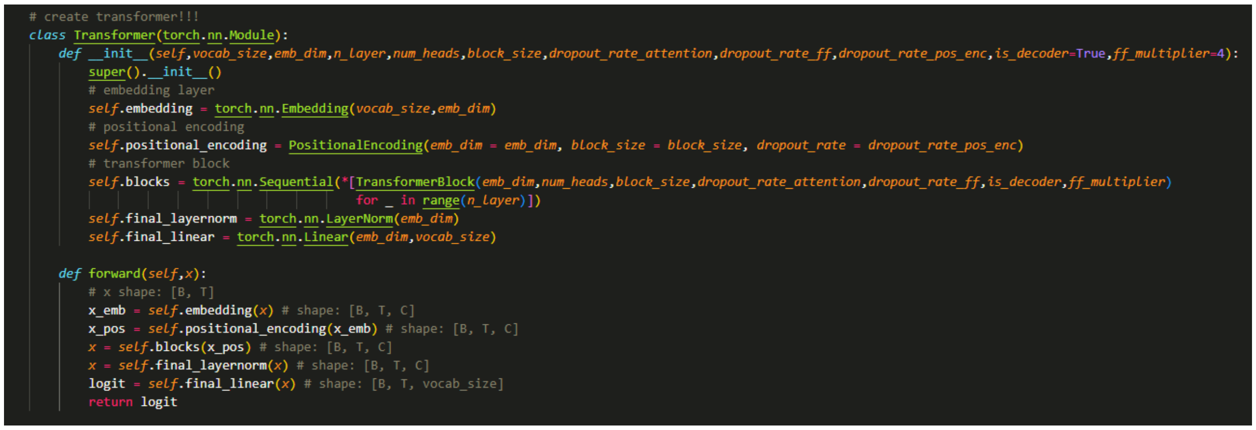
- Full code



```
# positional encoding from the paper
class PositionalEncoding(torch.nn.Module):
    def __init__(self, emb_dim, block_size,dropout_rate):
        super().__init__()
        self.dropout = torch.nn.dropout(dropout_rate)
        # create common divisers: 1 / n^(2i/emb_dim) where i = 0, ..., (emb_dim/2)-1
        #  with some manipulation, this can be exp(-2i*log(n)/emb_dim)
        div_term = torch.exp(-torch.arange(0, emb_dim, 2)*math.log(10000)/emb_dim) # shape (emb_dim/2,)
        # k = 0, ..., block_size-1, column vector
        k = torch.arange(block_size).unsqueeze(1) # shape [block_size,1]
        # create calculation between sin and cos
        product_term = k * div_term # shape [block_size,emb_dim/2]

        # initialize PE matrix
        pe = torch.zeros((block_size,emb_dim))
        # assign values
        pe[:,0::2] = torch.sin(product_term)
        pe[:,1::2] = torch.cos(product_term)
        # create batch dimension
        pe = pe.unsqueeze(0)

        # make the parameters untrainable
        self.register_buffer("pos_enc", pe)

    def forward(self, x):
        output = x + self.pos_enc
        return self.dropout(output)
```

Transformer Part 9: Transformer Class

- Wrape everything up



```python
# create transformer!!!
class Transformer(torch.nn.Module):
    def __init__(self,vocab_size,emb_dim,n_layer,num_heads,block_size,dropout_rate_attention,dropout_rate_ff,dropout_rate_pos_enc,is_decoder=True,ff_multiplier=4):
        super().__init__()
        # embedding layer
        self.embedding = torch.nn.Embedding(vocab_size,emb_dim)
        # positional encoding
        self.positional_encoding = PositionalEncoding(emb_dim = emb_dim, block_size = block_size, dropout_rate = dropout_rate_pos_enc)
        # transformer block
        self.blocks = torch.nn.Sequential(*[TransformerBlock(emb_dim,num_heads,block_size,dropout_rate_attention,dropout_rate_ff,is_decoder,ff_multiplier)
                                            for _ in range(n_layer)])
        self.final_layernorm = torch.nn.LayerNorm(emb_dim)
        self.final_linear = torch.nn.Linear(emb_dim,vocab_size)

    def forward(self,x):
        # x shape: [B, T]
        x_emb = self.embedding(x) # shape: [B, T, C]
        x_pos = self.positional_encoding(x_emb) # shape: [B, T, C]
        x = self.blocks(x_pos) # shape: [B, T, C]
        x = self.final_layernorm(x) # shape: [B, T, C]
        logit = self.final_linear(x) # shape: [B, T, vocab_size]
        return logit
```

Transformer Part 10: a leap forward to ChatGPT

- First stage: pre-train like above
- Second stage: fine tune
  - Step 1: collect training data in the format of question on top and answer below
  - Step 2: train a reward model
  - Step 3: use reinforcement learning to optimize a policy against the reward model

Ideas

Well trained language model make inference on new words

- Once the language model is well trained, it should have meaningful representations of words, if new word is encountered, we can ask it to search the definition, and run the definition through the model to produce a context vector that will become the embedding for the new word
- Intuitively, the model is learning new words just like us, by reading and understanding definitions of the new words