WORKING IN OWN VIRTUAL ENVIRONMENT WITH IPYKERNEL AS LEXAWEAVE

In [43]:
```
import torch.nn as nn
from torch.nn import functional as F
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
max_iters = 10000 #how many iterations we are going to have in the training loop
# eval_interval = 2500
learning_rate = 3e-4
eval_iters = 250
#dropout = 0.2 - not active in eval mode
#dropout - it is going to dropout random neurons in the network so that we dont ove
```

```
cpu
```

In [44]:
```
block_size = 8 # for length of integers
batch_size = 4 # for parallel processing
```

In [45]:
```
with open('wizard_of_oz.txt', 'r', encoding='utf-8') as f:
    text = f.read()
chars = sorted(set(text))
print(chars)
vocab_size = len(chars)
```

```
['\n', ' ', '!', '"', '&', "'", '(', ')', ',', '-', '.', '0', '1', '2', '3', '4',
'5', '6', '7', '8', '9', ':', ';', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'Y',
'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\ufeff']
```

In [46]:
```
string_to_int = { ch:i for i,ch in enumerate(chars) }
int_to_string = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])

data = torch.tensor(encode(text), dtype=torch.long)
## super long sequence of integers  --> dtype
# print(data[:100])
```

In [47]:
```
import torch
#torch handles a lot of math calculus and linear algebra
```

torch handles a lot of math calculus and linear algebra,

linear algebra has a data structure named tensors

PACKAGES INSTALLED : MATPLOTLIB, NUMPY, IPYKERNEL, PYLZMA, TORCH FROM PYTORCH DOCS - USING CUDA 11.8

EVERYTHING TO BE PUT INSIDE TENSOR SO ITS EASIER FOR PYTORCH TO WORK WITH

VALIDATION AND TRAINING SPLITS

BIGRAM MODEL USING ANN, taking a small snipet from the entire corpus of text and offset by one

In [48]:
```
## start of the snippet to block size 5
# x = predictions, y = targets
```

blocks stacked on top of each other - batch size

Scaling large language models

block size : length of each sequence batch size : how many of these are we doing at the same time

Bigram model : doesn't know the history of knowledge but predicts the next character based on what the current character is

from starting point of indice to the length of the data - will be the range

random indices in the entire text that we can start generating from

In [49]:
```
n = int(0.8*len(data))
train_data = data[:n]
val_data = data[n:]

def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix]) # stacks them in batches
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y

x, y = get_batch('train')
print('inputs:')
# print(x.shape)
print(x)
print('targets:')
print(y)
```

```
inputs:
tensor([[43, 56, 49, 68,  1, 57, 67,  1],
        [54, 68, 53, 66,  1, 49, 60, 60],
        [64,  1, 63, 54,  1, 30, 60, 57],
        [55,  1, 33, 57, 61,  8,  1, 71]])
targets:
tensor([[56, 49, 68,  1, 57, 67,  1, 65],
        [68, 53, 66,  1, 49, 60, 60, 10],
        [ 1, 63, 54,  1, 30, 60, 57, 62],
        [ 1, 33, 57, 61,  8,  1, 71, 56]])
```

Initialising our neural net:

FOR REPORTING LOSSES, GRADIENTS ARE USED WHEN TRAINING IS ON; BUT WHEN IT IS IN TESTING, GRADIENTS ARE OFF

In [60]:
```
@torch.no_grad() #it makes sure pytorch doesn't use gradients at all in here
#that'll reduce computation

def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

https://www.wolframalpha.com/

small example: for calculating loss --> //playing with this : gradient descent

loss is measured by taking negative log likelihood

lets say we have 80 characters in our vocabulary, and we have just started our model,

no training at all, just completely random weights,

then there is a 1 in 80 chance that we predict next character successfully

we will be using Adam w optimiser:

Adam uses the moving average of both the gradient and its squared to adapt the learning rate of each parameter

adam w - adds weight decay - means it generalises the parameters more, prevents it from having super high performance or super low performance, generalises it in between

,we need a small learning rate

preparing an embedding table

if we use the nn module - then its going to be a learnable parameter

Using forward pass and predicting what character comes next using logits

To understand all the transformations, architectures going on behind the scenes,like getting an input , running it through a network and getting an output ,also for flexibility debugging and optimisation

nn.embedding - is like a lookup table ,it is a giant sort of grid of what the predictions are going to look like

normalising - means how significant is that to the entire row ,so if suppose 'rl' has a number of 40000 and 'ab' has 300, 'bc' has 4000 ,then rl has a fairly high prob of coming next , means a lot of times you are going to have an l coming after an r, this is the meaning.

normalising - taking the contribution of an element to the sum of all elements

logits - bunch of floating point numbers that are normalised Ex: [2,4,6] => 2/12, 4/12, 6/12 => [0.167,0.33,0.5] - logits logits - are like a prob distribution ==> lets assign ab =0.167, ac=0.33, ad =0.5 so there is a high prob that a is followed by d (high prob)

B : Batch; time dimension(T) : sequence of integers, WE start from here and we dont know what the next token is ? basically because there's some we dont know yet and there's some we already know; Channel (C) : is the vocabulary size - we can have different channels - main attention is given to the channels So we will blend the Batch and time together (B*T)

LOGITS AND TARGETS SHOULD HAVE THE SAME BATCH IN TIME

pytorch expects the shape to be in the format of (N,C) -- is like (B*T,C)

In [61]:
```
class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, index, targets=None):
        logits = self.token_embedding_table(index)


        if targets is None: #RETURNS 3DIM LOGITS
            loss = None
        else: # RETURNS 2DIM LOGITS
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

GENERATING TOKENS

In [62]:
```
    def generate(self, index, max_new_tokens):
        # index is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self.forward(index) #BASED ON CURRENT STATE OF MODEL
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
```

```
            # sample from the distribution
            index_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            index = torch.cat((index, index_next), dim=1) # (B, T+1)
            # CONCATENATING MORE TOKENS TO IT [CURRENT + 1 = NEXT TOKEN]
        return index




## NEGATIVE INDEXING : IF YOU GO BEFORE 0, ITS GOING TO LOOP TO THE VERY END OF THA
## -1 : LAST ELEMENT OF THE ARRAY => LAST DIMENSION
## -2 : SECOND LAST ELEMENT OF THE ARRAY
## -3 : THIRD LAST ELEMENT OF THE ARRAY
```

In [63]:

```
class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, index, targets=None):
        logits = self.token_embedding_table(index)


        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, index, max_new_tokens):
        # index is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self.forward(index)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            index_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            index = torch.cat((index, index_next), dim=1) # (B, T+1)
        return index

model = BigramLanguageModel(vocab_size)
m = model.to(device)

context = torch.zeros((1,1), dtype=torch.long, device=device)
generated_chars = decode(m.generate(context, max_new_tokens=500)[0].tolist())
print(generated_chars)
```

```
PbShYIDYo6!qtj!2GAD4Ua,HN)JBc(3y)g?TYWyk&Tyi.8iP.STSwKFJ7z(NMteD!vWKid
3g(z&MZ:.D3xN:xO!Nf IkGlgIhjojGTODigtLC1:JSGIO&8yOwTvE8-h(!jvLU3E1:lgub;:
L!wU?SnSG4;yQ'G435scGo
aqf!kxPZGMplm21inndc7., J1"HyPKeqAnsGoM'Z4Zs5cOZTN5rqhENeJ3?k?dUkgdN:08Hygde9GoEa
5r-1T)okVJ
.ZGzq;r?;8JqfrA)Ji;)Wuhz UD?ySJd"kgGA)OvNTyjPhI'OcT,zTMGo6l.1:U?Mb8dDNMtJ6 )rvWyte
I&PZtQD)Fz(p:09j."KysLWyIV 5ddH5k(FG1V-"FOY8z( HdKytC!e.m,f&yrDFidwnsPbd5g8z(Bb:bg
lz'x!7MKUbo6lpW()9iRm!&O9TCG1"N:lAdlK,Fz(K-0uW96?(C)CC:Jk!pW?9uIMpBN-
```

TRAINING LOOP

In [64]:
```
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

#standard training loop architecture
# --------------------------------------------------------------------------------
for iter in range(max_iters):
    if iter % eval_iters == 0:
        losses = estimate_loss()
        print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses[

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model.forward(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    #by default pytorch will accumulate the gradients over time via adding them
    #by putting zero grad we make sure they do not add over time
    #so the previous gradients dont affect the current one
    #previous gradients are from previous data
    #data is kind of weird sometimes, sometimes its biased and we dont want that de
    # like what our error is right ?
    #so we only want to decide, only want to optimise based on the current gradient
    loss.backward()
    optimizer.step() #lets gradient descent work # step in the right direction
print(loss.item())
#--------------------------------------------------------------------------------
```

```
step: 0, train loss: 4.694, val loss: 4.702
step: 250, train loss: 4.622, val loss: 4.648
step: 500, train loss: 4.573, val loss: 4.573
step: 750, train loss: 4.527, val loss: 4.494
step: 1000, train loss: 4.456, val loss: 4.466
step: 1250, train loss: 4.404, val loss: 4.395
step: 1500, train loss: 4.319, val loss: 4.328
step: 1750, train loss: 4.284, val loss: 4.307
step: 2000, train loss: 4.237, val loss: 4.242
step: 2250, train loss: 4.190, val loss: 4.176
step: 2500, train loss: 4.145, val loss: 4.132
step: 2750, train loss: 4.090, val loss: 4.085
step: 3000, train loss: 4.039, val loss: 4.041
step: 3250, train loss: 3.984, val loss: 4.002
step: 3500, train loss: 3.961, val loss: 3.935
step: 3750, train loss: 3.904, val loss: 3.905
step: 4000, train loss: 3.862, val loss: 3.872
step: 4250, train loss: 3.813, val loss: 3.826
step: 4500, train loss: 3.777, val loss: 3.785
step: 4750, train loss: 3.739, val loss: 3.736
step: 5000, train loss: 3.699, val loss: 3.711
step: 5250, train loss: 3.653, val loss: 3.660
step: 5500, train loss: 3.612, val loss: 3.606
step: 5750, train loss: 3.583, val loss: 3.586
step: 6000, train loss: 3.541, val loss: 3.579
step: 6250, train loss: 3.508, val loss: 3.524
step: 6500, train loss: 3.476, val loss: 3.493
step: 6750, train loss: 3.430, val loss: 3.456
step: 7000, train loss: 3.407, val loss: 3.421
step: 7250, train loss: 3.400, val loss: 3.408
step: 7500, train loss: 3.350, val loss: 3.381
step: 7750, train loss: 3.307, val loss: 3.320
step: 8000, train loss: 3.287, val loss: 3.299
step: 8250, train loss: 3.263, val loss: 3.250
step: 8500, train loss: 3.231, val loss: 3.241
step: 8750, train loss: 3.210, val loss: 3.221
step: 9000, train loss: 3.184, val loss: 3.197
step: 9250, train loss: 3.159, val loss: 3.192
step: 9500, train loss: 3.109, val loss: 3.149
step: 9750, train loss: 3.084, val loss: 3.120
2.8330066204071045
```

In [65]:

```
context = torch.zeros((1,1), dtype=torch.long, device=device)
generated_chars = decode(m.generate(context, max_new_tokens=500)[0].tolist())
print(generated_chars)
```

```
ZCccij7n tK4l ZIODFAI6CNebQM0OncTQz(w?OusoD?G"VpFu H6;gu
"Ez!!, bqjGwing.!,Yis-V; -D
xb
Ye t d
.;
"3Hon4fBC&!5h es t HJ3noe TcinvJc440
.Zser93
avsopRF"8q3&Ag2-""Et6
59ideexRYjur Ha 5r Btzz
E8;42vnRN3?)JpCualpee,m;6TY8zwn aQGon-0upulflore wh4;
Bid,h:AqV;ee and
drHHHu an"H.moAd.nGof-0M8yer M!G9iRr,xh bo H;eyw,'mT,he,ongh:-vsequg.xMThisn.
"HpWs cl tho-!F1 yw, olfCEEn,gdid:Et sed
Ps"HE6,;mwn l,yLSB
y:3lTNBqPYft,s,xiounkej'zqIDfe'Lkdp9Minw)I5Drsend sw
A)Rd
aplCl,y lyr-qYJUeKhlKfx!)Ich awA?)dore t
```

ACTIVATION FUNCTIONS

RELU, SIGMOID, TANH

RELU - if a number is 0 or below 0, it gets converted to 0, if a number is above 0, it will stay the same

```
- it simply offers non linearity to our linear networks
```

SIGMOID - outputs values between 0 and 1, TANH - outputs values between -1 and 1

In [67]:
```
x = torch.tensor([-0.05], dtype=torch.float32)
# its basically 1/1+exp(-x)
y = F.sigmoid(x)
print(y)
```

```
tensor([0.4875])
```

In [ ]: