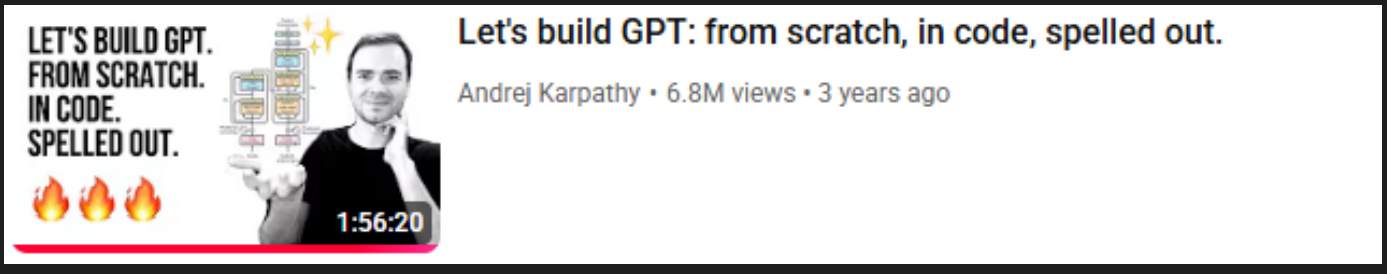
companion notes for 'Let's build GPT'

lecture by Anderj Karpathy



Function to create batch:

```python
batch_size = 4 #how many independent sequence will we process in parallel
block_size = 8 #what is the maximum context length for prediction

def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    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])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return x,y
```

switch between train/text data

subtracting the block_data so the selection window won't go out of bounds.

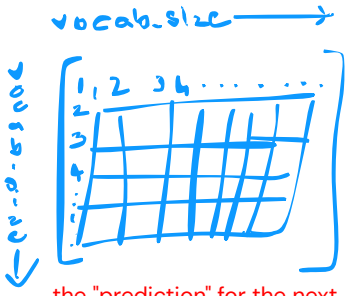
`torch.randint(low, high, size)`

creating an vector of random samples(starting index)

stacking the data and creating the 'query' and 'response' (- label) matrices for training

Bi-gram model:

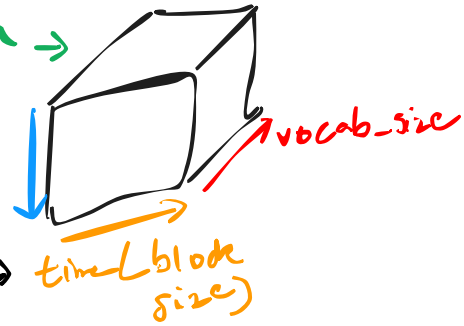
```python
import torch.nn as nn
from torch.nn import functional as F
torch.manual_seed(1337)

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits  for the next token from a lookup table.
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets):
        #idx and targets are both (B, T) tensor of integers
        logits = self.token_embedding_table(idx) # (B, T, C)

        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

m = BigramLanguageModel(vocab_size)
logits, loss = m(xb, yb)
print(logits.shape)
print(loss)
```



the "prediction" for the next character is just a single row in this table. If you are at character S, you go to the S row in the table, and the numbers in that row tell you how likely every other character is to come next.

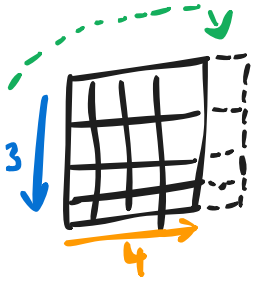
reshaping in order to fit the cross_entropy function inputs properly

```
torch.Size([4, 8])
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
```



Prediction:

```python
def generate(self, idx, max_new_tokens):
    # idx is (B, T) array of indices in current context
    for _ in range(max_new_tokens):
        # get the predictions
        logits, loss = self(idx)
        # 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
        idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
        # append the sampled index to the index sequence
        idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
    return idx
```



concat the sampled index to the sequence

Estimating loss:

```python
@torch.no_grad()
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
```

informing the model that we are going into evaluation mode(it is general practice to have certain architectural changes when evaluating the model)

switching back to training mode

some linear algebra(masking):

```
tril
[24] ✓ 0.0s
tensor([[1., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1., 1., 1., 1.]])
```

lower triangular matrix. having the 'future tokens' to be zero - preventing the model from looking into the future

```
wei
[26] ✓ 0.0s
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]])
```

a sample weight matrix after applying softmax

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Numerator (e^{z_i}): This is the **Exponential**. It ensures every value is positive (since $e^x > 0$ for all real x). It also magnifies the differences between numbers (e.g., $e^3 \approx 20$, while $e^4 \approx 54$).

Denominator ($\sum e^{z_j}$): This is the **Normalizer**. By dividing the individual score by the sum of all scores, we guarantee the results sum to 1.

token vs positional embedding: (why not just assign index manually?)

- Token Embedding: "What is this character?" (e.g., 'a', 'b', 'c')
- Positional Embedding: "Where am I in the sequence?" (e.g., Index 0, 1, 2... up to block_size-1)

x = tok_emb + pos_emb

- Because we add the vectors, every single number that goes into the model now contains two pieces of information:
- "I am the letter 'E'..." + "...and I am currently at the 5th position in this block."

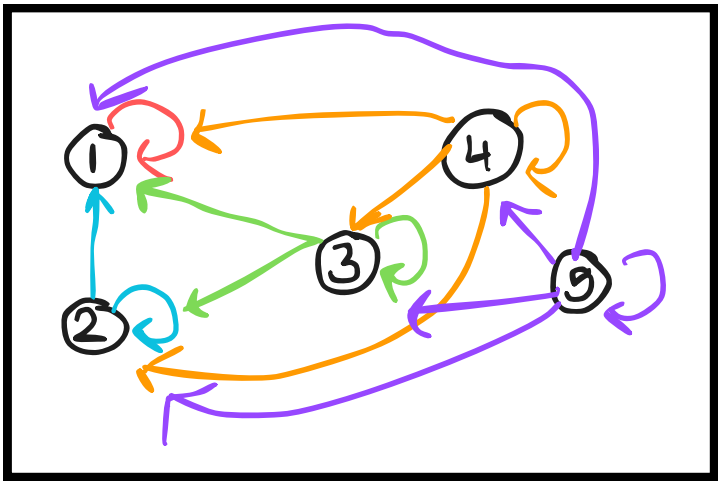
By using an embedding table, we give the model 65 (or C) trainable parameters to describe the "feature"/ "character" of being at index 5

self-attention:

what am i looking for?

What info do i contain?

Each token has a query vector and a key vector. The query vector of a token is compared(dot-product) with every other key vector and their 'compatibility' is quantified into the weights



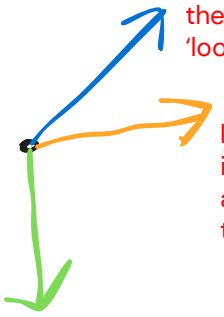
every node is affected by itself and the nodes that comes prior to it.

ITS JUST A COMMUNICATION FACILITATING THE TOKENS TO LEARN MORE ABOUT THEMSELVES AND THEIR PLACE IN THE LARGER CONTEXT

All these are still being done on a strictly independent basis. None of the tokens know where they are. The only way for them to know the positional value is through their respective positional embedding

FUNDAMENTAL DIFFERENCE BETWEEN ENCODER AND DECODER is that in the encoder attention block the tokens are allowed to 'see' the tokens from the future and that is not the case in the decoder block

**the above directed graph is a conceptual representation of the decoder self-attention block*



query vector- the information the token is 'looking' for

key vector- the information is answer the token holds

together form the weights(dot products)

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

value vector- the actual information that will be used once the weight matrix is calculated using the query-key vectors

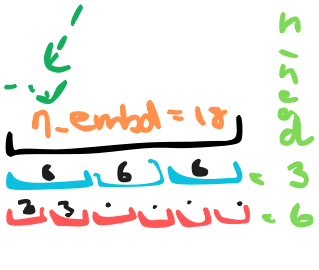
weights are multiplied with the values

multiple attention heads and head size

each head needs to be a fraction of the total embedding size so that we *cat* them back together, to end up with the same size we started with.

If there is x number of embeddings for your token - the number of attention head is y then the head size for each attention head must be x/y

```
class Block(nn.Module):
    """Transformer block: Communication followed by computation"""

    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(n_embd)

    def forward(self, x):
        x = x + self.sa(x)
        x = x + self.ffwd(x)
        return x
```



feed forward layer:

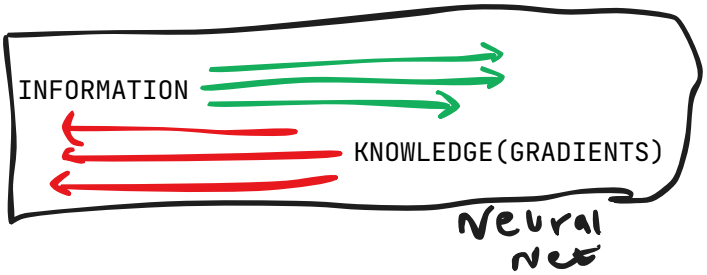
```python
class FeedForward(nn.Module):
    """A simple linear layer followed by a non-linearity"""

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, n_embd),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.net(x)
```

communication between nodes/tokens for learning information about themselves and each other - learning inter-dependencies

Attention - COMMUNICATION
Feed forward - COMPUTATION

computations performed by the individual nodes independently in the aftermath

vanishing gradient and residual connection:



the deeper the network the more 'distance' the gradients have to travel to reach their destination

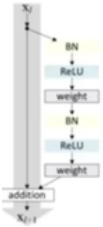
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial \text{layer}_n} \cdots \frac{\partial \text{layer}_2}{\partial \text{layer}_1} \cdot \frac{\partial \text{layer}_1}{\partial w_1}$$

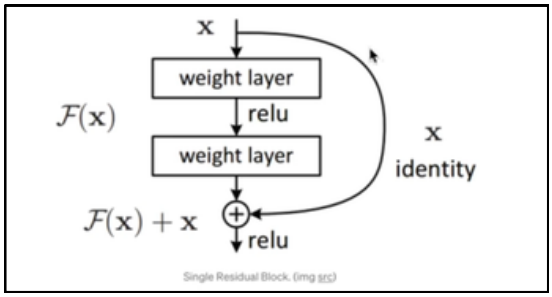
Imagine you have 10 layers. If the gradient at each layer is 0.1, the gradient at the start is 0.1^{10}, which is 0.0000000001

A Residual Connection (also known as a Skip Connection) is an architectural shortcut in a neural network that allows the input of a layer to "skip" the processing steps and be added directly to the output of that layer.

```
x = x + attention
```

$$\frac{dy}{dx} = 1 + F'(x)$$





layer norm:

is basically seeing how does each channel compare to the other within a specific token

Layer Normalization (LayerNorm):

$$y = \gamma \left(\frac{x - \mu}{\sqrt{\text{Var}(x) + \varepsilon}} \right) + \beta$$

Learnable parameters: γ, β

squashing all these numbers so they have a Mean of 0 and a Standard Deviation of 1

scaling and generating:

After wider embeddings, multiple heads, more layers(increased depth), increased context()block_size, VIOLA! We can generate gibberish in Shakespearean english

```python
# hyperparameters
batch_size = 64 # How many independent sequences will be
block_size = 256 # What is the maximum context length for
max_iters = 5000
eval_interval = 500
learning_rate = 3e-3
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_layer = 6
n_head = 6
dropout = 0.2
# ------------
```

```
(.venv) PS D:\GPT from scratch> python bigram_v2.py
step 0: train loss 4.4753, val loss 4.4709
step 500: train loss 1.9285, val loss 2.0308
step 1000: train loss 1.5402, val loss 1.7343
step 1500: train loss 1.4032, val loss 1.6396
step 2000: train loss 1.3180, val loss 1.6032
step 2500: train loss 1.2512, val loss 1.5906
step 3000: train loss 1.2006, val loss 1.5850
step 3500: train loss 1.1358, val loss 1.6419
step 4000: train loss 1.0824, val loss 1.6546
step 4500: train loss 1.0163, val loss 1.7527
Generated 501 tokens.

BRAKENBURY:
I ammpty your grace will leave a luckay.
What wine have any distressed corpets scarce in them?
And who determine his you made pan these shoon?
Forgive my brother wrong'd he of these bed;
And, for his beauty two; were wild you vit,
That, with him as you.

Justicer:
Henry, on my affair.

Fourth!
OR
ANIUS:
Who's! one, Julius make not thence breathes;
And, herethinks, I mean her it made my shapes brife.

LARTIUS:
Well begun. Farewell.

COMINIUS:
Mark you specially know his course will no
```