# Self Attention Notes

One thing we want to do now is to better approximate $\mathbb{P}(x_1, x_2, \ldots, x_n)$. The goal of this notebook is to motivate and familiarize you with the mechanism that powers transformers: self-attention. But before we do that, we need to understand what *embeddings* are.

## Embeddings / Representations

In the neural networks context, embeddings are vectors which represent a data point or piece of information. In the NLP case, we can think of embeddings as being "word vectors" or "token vectors." In the context of this project, we will assign tokens integer IDs. I.e. the letter "a" might be assigned to ID 0. We can think of an "embedding table" as a function that maps these integers to n-dimensional vectors, i.e.

$$f : \mathbb{Z}^+ \to \mathbb{R}^n$$

Typically, **the embeddings are much higher dimensional than the vocabulary**. This is important as we'll see later. Quick aside: I'll generally use the terms embeddings and representations interchangably. But the *embeddings layer* is the embedding lookup table that creates the initial representation.

In [121…
```
import numpy as np

np.random.seed(42)
alphabet = "abcdefghijklmnopqrstuvwxyz ."

vocab_size = len(alphabet)
embed_dim = 512
txt2idx = {alphabet[i]:i for i in range(len(alphabet))}
idx2txt = {i:alphabet[i] for i in range(len(alphabet))}

embedding_table = np.random.normal(size=(vocab_size, embed_dim))
def embed_token(token):
    token_idx = txt2idx[token]
    return embedding_table[token_idx]

def embed_sentence(sentence):
    out = []
    for t in sentence:
        out.append(
            embed_token(t)
        )
    return np.array(out)

embeddings = embed_sentence("hey sean. whats up")
print(embeddings.shape)
```

```
(18, 512)
```

By converting

- text -> tokens
- tokens -> integers
- integers -> embeddings

we effectively have a function that can map a sentence to a sequence of vectors.

## Aside on embedding size and information:

If we have a vocabulary of ~64k, we need ~16 bits = 2 bytes of information to express each word in our vocabulary. Or, in other word, each word can be expressed by a 16-dimensional one-hot vector. By using embeddings (say at fp16 precision and 512 dimensional), each word goes from needing 2 bytes to represent to needing 1024 bytes of information. One reason we require more bytes is because of the higher dimensionality, and the other is the switch from a discrete representation to a continuous representation.

Why do we do this? Why do we represent words in neural networks with so much redundancy? Consider all the cases where the word "the" might be used. In one sentence the word "the" can take on multiple contexts -- ideally, the "true" embeddings of a token or word (or character) should correspond to a **contextualized** representation of that word. So choosing to represent words as high dimensional vectors gives our model a lot of room to change the vector when contextualized.

# How do we use representations for language modeling?

A natural question that arises is: how are embeddings useful in practice? I.e. how do they work within a Transformer, and how do they contribute to learning $\mathbb{P}$?

Consider the below code:

In [121…
```
out_proj = np.random.normal(size=(embed_dim, vocab_size))

embeddings[-1] @ out_proj
```

Out[1216]:
```
array([  7.7038926 , -23.38859578,   4.08575033,  32.75940082,
        -2.46142945,   1.17977083,   2.80594505, -19.73685707,
        -5.67724779, -12.9283562 ,  -5.82059853,  14.05219237,
        -6.69960456,  19.01724971,   4.13239048,  16.48436548,
         8.10346048,  33.49901703,  18.96133537,   7.04676268,
         1.37820658,  24.86854581,  -6.82997835,  29.901247  ,
        27.4442641 , -24.14996626, -19.31185507,  34.02881432])
```

What I've constructed above is a **projection matrix**. This will take an embedding, and *project* it to another vector space -- in this case, back to the embedding dim. The idea behind these embeddings and internal representations is that, if they're good enough, we can project them back to the vocab space as a distribution over the next token. E.g. as part of our transformer language model, the final layer will be a matrix

$$M : \mathbb{R}^d \to \mathbb{R}^{\text{vocab}}$$

which then gives us our probability distribution, and our language model. In other words, the goal of a Transformer is to learn good enough representations such that the correct next-token distribution $\mathbb{P}$ is a linear function of our representations. In practice, we typically just project the *last* token in our sequence to get this, and I'll explain why that is shortly. Keep in mind that in order to ensure the output of the projection is a true distribution, we use something called *softmax* to normalize the distribution to add up to 1.

Now currently, both our embedding table and projection matrix aren't particularly useful -- they're both randomly initialized. But the point is that with enough data, we can update these such that they work to make useful predictions.

## The Training Objective

How do neural networks learn from data? In the simplest terms, they see tons of examples, and use something called a **loss function** to tell the network how good its predictions were. It then uses calculus to figure out how to minimize this loss function by updating its weights. In the above case, our simple neural network would be updating the embedding table and the projection matrix to minimize the loss.

So what loss do Language Models use? They use something called **cross entropy**:

$$L = \frac{1}{N} \sum_{i=1}^{N} -ln(y_i)$$

with $y_i$ being the **likelihood of the correct token as predicted by the model**.

For example, if my language model assigns the token "j" a likelihood of $0.03$ but it was the correct token given the context, we would say that the loss for that example is $-ln(0.03) \approx 3.5$. Minimizing this loss corresponds with maximizing the likelihood of the correct token $y_i$.

Using the embedding table, proj matrix, softmax, and cross entropy, I'll show a brief implementation of training this rudimentary network in pytorch using the tiny shakespeare dataset. This isn't the full Transformer (it's actually something we've covered before!), but this is the skeleton of what mechanistically makes it work. After this section we'll discuss the crux of this lecture: self-attention.

In [121…

```
import torch
from torch import nn
import torch.nn.functional as F

# lots of code adapted from the goat Andrej karpathy
```

```
torch.manual_seed(42)

with open('tinyshakespeare.txt', 'r') as f:
    text = f.read()

chars = sorted(list(set(text)))
vocab_size = len(chars)
print(f'Vocab: {chars}')
print(f'Vocab size: {vocab_size}')
```

```
Vocab: ['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?', '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', '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']
Vocab size: 65
```

In [121…

```
# Prepare the dataset
# create a mapping from characters to integers
txt2idx = { ch:i for i,ch in enumerate(chars) }
idx2txt = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [txt2idx[c] for c in s] # encoder: take a string, output a
decode = lambda l: ''.join([idx2txt[i] for i in l]) # decoder: take a list of

print(encode("hi there!"))
print(decode(encode("hi there!")))
```

```
[46, 47, 1, 58, 46, 43, 56, 43, 2]
hi there!
```

In [122…

```
class SoftmaxLM(nn.Module):
    def __init__(self, vocab_size, channel_dim=64):
        super().__init__()
        # each token directly reads off the logits for the next token from a l
        self.token_embedding_table = nn.Embedding(vocab_size, channel_dim)
        self.proj = nn.Linear(channel_dim, vocab_size)

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.proj(self.token_embedding_table(
            idx
        )) # (B, T, C)

        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, idx, max_new_tokens):
        # idx is (B, T) array of indices in the 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 sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

In [122…

```
data = torch.tensor(encode(text), dtype=torch.long)

n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]

block_size = 8
print(f'Example sample: {train_data[:block_size+1]}')

x = train_data[:block_size]
y = train_data[1:block_size+1]
for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f"when input is {context} the target: {target}")
```

```
Example sample: tensor([18, 47, 56, 57, 58,  1, 15, 47, 58])
when input is tensor([18]) the target: 47
when input is tensor([18, 47]) the target: 56
when input is tensor([18, 47, 56]) the target: 57
when input is tensor([18, 47, 56, 57]) the target: 58
when input is tensor([18, 47, 56, 57, 58]) the target: 1
when input is tensor([18, 47, 56, 57, 58,  1]) the target: 15
when input is tensor([18, 47, 56, 57, 58,  1, 15]) the target: 47
when input is tensor([18, 47, 56, 57, 58,  1, 15, 47]) the target: 58
```

In [122…

```
torch.manual_seed(1337)
batch_size = 4 # how many independent sequences will we process in parallel?
block_size = 8 # what is the maximum context length for predictions?

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

xb, yb = get_batch('train')
print('inputs:')
print(xb.shape)
print(xb)
print('targets:')
print(yb.shape)
print(yb)
```

```
print('----')

for b in range(batch_size): # batch dimension
    for t in range(block_size): # time dimension
        context = xb[b, :t+1]
        target = yb[b,t]
        print(f"when input is {context.tolist()} the target: {target}")
```

```
inputs:
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]])
targets:
torch.Size([4, 8])
tensor([[43, 58,  5, 57,  1, 46, 43, 39],
        [53, 56,  1, 58, 46, 39, 58,  1],
        [58,  1, 58, 46, 39, 58,  1, 46],
        [17, 27, 10,  0, 21,  1, 54, 39]])
----
when input is [24] the target: 43
when input is [24, 43] the target: 58
when input is [24, 43, 58] the target: 5
when input is [24, 43, 58, 5] the target: 57
when input is [24, 43, 58, 5, 57] the target: 1
when input is [24, 43, 58, 5, 57, 1] the target: 46
when input is [24, 43, 58, 5, 57, 1, 46] the target: 43
when input is [24, 43, 58, 5, 57, 1, 46, 43] the target: 39
when input is [44] the target: 53
when input is [44, 53] the target: 56
when input is [44, 53, 56] the target: 1
when input is [44, 53, 56, 1] the target: 58
when input is [44, 53, 56, 1, 58] the target: 46
when input is [44, 53, 56, 1, 58, 46] the target: 39
when input is [44, 53, 56, 1, 58, 46, 39] the target: 58
when input is [44, 53, 56, 1, 58, 46, 39, 58] the target: 1
when input is [52] the target: 58
when input is [52, 58] the target: 1
when input is [52, 58, 1] the target: 58
when input is [52, 58, 1, 58] the target: 46
when input is [52, 58, 1, 58, 46] the target: 39
when input is [52, 58, 1, 58, 46, 39] the target: 58
when input is [52, 58, 1, 58, 46, 39, 58] the target: 1
when input is [52, 58, 1, 58, 46, 39, 58, 1] the target: 46
when input is [25] the target: 17
when input is [25, 17] the target: 27
when input is [25, 17, 27] the target: 10
when input is [25, 17, 27, 10] the target: 0
when input is [25, 17, 27, 10, 0] the target: 21
when input is [25, 17, 27, 10, 0, 21] the target: 1
when input is [25, 17, 27, 10, 0, 21, 1] the target: 54
when input is [25, 17, 27, 10, 0, 21, 1, 54] the target: 39
```

In [122…

```
print(xb) # our input to the transformer
```

```
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]])
```

In [122…
```
m = SoftmaxLM(vocab_size)
logits, loss = m(xb, yb)
print(logits.shape)
print(loss)

print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_t
```

```
torch.Size([32, 65])
tensor(4.3428, grad_fn=<NllLossBackward0>)

Z
QiewEpPV.mLoRqApmSnAhXVK
$hR
fYkdFBUy-aMv,ieetsrJbc3k'ALxidKUy;RpPqAyHvs:TorvT?rVUC,he$
NjeAAzX$tG
```

In [122…
```
batch_size = 128
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
for steps in range(1000): # increase number of steps for good results...

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

    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.4928877353668213
```

In [122…
```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_t
```

```
A beoteamaill vousivecome nd by we apre--hocoongs me woucathey ifomexe layimei
me: me thiatld, ps brerlitw'te. sids as ntinor wist. k hods, olde. dicenmerd E
Tou'd sout is.
POrd beid fely owhte Thillles;
core arwiefendind y t? mon bemanghirimy dime, thorarslf t.
FFoumper of trse ttou, bad ght
I my d!

HWou ngremer at wo heaco f, wiventherieretevono tr as lyon le o!
l s,
Y pe'st
Tr; mo-
I ofamorltthace weand thother than tis sthd hosot.
VI in hive G cho jufe o; t y atee southasl's? ind chivous dous
```

These results are pretty cool! But one problem with this current model is that **it doesn't contextualize anything** -- it simply takes the last token embedding and uses it to predict. In a sense, we've built a type of bigram with gradient descent instead of counting.

So how do we make this better? The answer is context.

# Contextualizing our Language Model

The missing piece from our current setup is that tokens don't *interact* with each other. In other words, our model doesn't take context into account. There is a static embedding table, and the embeddings of other things in the context dont influence the output. So now I'll try to motivate attention by discussing context.

## Averaging

To start, perhaps the simplest way to bake context into our model is by simply averaging all the embedding vectors before projecting. Currently, we simply pluck out the last embedding vector and project, but we can instead average out the sequence and then project the average. Let's train that and see how it does.

In [122…

```
class SimpleLMWithAveraging(nn.Module):

    def __init__(self, vocab_size, channel_dim=64):
        super().__init__()
        # each token directly reads off the logits for the next token from a l
        self.token_embedding_table = nn.Embedding(vocab_size, channel_dim)
        self.proj = nn.Linear(channel_dim, vocab_size)
        self.channel_dim = channel_dim

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        x = self.token_embedding_table(idx) # (B,T,C)
        xbar = x.mean(axis=1, keepdims=True).repeat(1, T, 1) # (B, 1, C) -> (B,

        logits = self.proj(xbar) # (B,T,vocab_size)

        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, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
```

```
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # 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)
            # print(probs)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
            # print(idx)
        return idx
```

In [123…
```
m = SimpleLMWithAveraging(vocab_size)
```

In [123…
```
batch_size = 128
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
for steps in range(1000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')
    # print(xb.shape)
    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.3625059127807617
```

In [123…
```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_t
```

U

Quite sus unfortunately. We still have some issues here:

- all averages all the same -- when we average and replace all the embeddings with the average over the sequence, we essentially predict the same thing for each token. this is bad.

- tokens looking ahead -- the token at the beginning of the sequence averages with tokens after it. this is bad, as when we're generating, each token can't look past itself. So we need to change the averaging code slightly to ensure each vector only averages with things before it.
- positional embeddings -- currently each word gets an embedding, but we don't have a way to indicate to the model which timestep each token is currently at. we can add this by just adding a small amount to each vector depending on its position
- context-dependent averaging -- ideally, the average we compute should be *dependent on the token we're looking at*. We can try computing the dot product between each embedding and each other embedding -- this will allow the model to selectively decide weights for each element of the sequence

Let's implement some architectural changes to alleviate this. The next iteration will have the following key features:

- no token lookahead with *masking*
- positional embeddings with another embedding table
- context-dependent averaging with dot products

## Quick aside: matmul as masking

Let's say we have some tensors of the shape

In [988…
```
R, C = 3, 3
x = torch.randint(0,10,(R, C)).float()
x
```

Out[988]:
```
tensor([[6., 0., 9.],
        [3., 3., 1.],
        [3., 8., 9.]])
```

If we want to average across the columns, and replace all the values, we can just compute

In [990…
```
x.mean(axis=0).repeat(R, 1)
```

Out[990]:
```
tensor([[4.0000, 3.6667, 6.3333],
        [4.0000, 3.6667, 6.3333],
        [4.0000, 3.6667, 6.3333]])
```

But another way to do this is with matrix multiplication! Consider the following matmul:

In [991…
```
w = torch.ones(R, C)
w = w / torch.sum(w, 1, keepdim=True)
w
```

Out[991]:
```
tensor([[0.3333, 0.3333, 0.3333],
        [0.3333, 0.3333, 0.3333],
        [0.3333, 0.3333, 0.3333]])
```

In [992…
```
w @ x # matrix mul
```

Out[992]:
```
tensor([[4.0000, 3.6667, 6.3333],
        [4.0000, 3.6667, 6.3333],
        [4.0000, 3.6667, 6.3333]])
```

Very cool! We can actually use this to our advantage though. Now consider the following weight matrix

In [994…
```
w = torch.tril(torch.ones(R, C))
w = w / torch.sum(w, 1, keepdim=True)
w
```

Out[994]:
```
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
```

When we compute the product

In [996…
```
print(f'original mat: {x}')
print(f'no lookahead avg: {w @ x}') # matrix mul
```

```
original mat: tensor([[6., 0., 9.],
        [3., 3., 1.],
        [3., 8., 9.]])
no lookahead avg: tensor([[6.0000, 0.0000, 9.0000],
        [4.5000, 1.5000, 5.0000],
        [4.0000, 3.6667, 6.3333]])
```

We get the averaging we want without a lookahead! The first row is the same as in the original matrix, since we only average 1 thing. But in subsequent rows, our average only takes into account values in each column that's been previously encountered. We can use this trick to stop the averaging lookahead!

In [123…
```
class SimpleLMWithBetterAveragingAndPosEmb(nn.Module):

    def __init__(self, vocab_size, seqlen=8, channel_dim=64):
        super().__init__()
        # each token directly reads off the logits for the next token from a l
        self.token_embedding_table = nn.Embedding(vocab_size, channel_dim)
        self.position_embedding_table = nn.Embedding(block_size, channel_dim)
        self.weights = nn.Linear(channel_dim, seqlen)
        self.proj = nn.Linear(channel_dim, vocab_size)
        self.channel_dim = channel_dim

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # we do positional embeddings now.
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)

        # no lookahead trick + weighted average
        wei = x @ x.transpose(1, 2) # compute weights for a weighted average b
```

```
        tril = torch.tril(torch.ones(T, T))  # remove weights for elements past
        wei = wei.masked_fill(tril == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1) # normalize weights to sum to 1

        xbar = wei @ x
        logits = self.proj(xbar) # (B,T,vocab_size)

        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, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # 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 sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

m = SimpleLMWithBetterAveragingAndPosEmb(vocab_size)
```

In [123…

```
batch_size = 128

optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
for steps in range(1000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')
    # print(xb.shape)
    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.4606504440307617
```

In [123…

```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_t
```

```
Nowsigng we f ire s d


GTir:

Wem, cetsou te w h sho a othorirer mefrere n stomullouetheipendlit tend he bes
e,
Bllos I nk noke iendy pir deit poney t, shoind y, win isploy, y ache Thd otor
y, fonden he rtly to wor;
Whithif her a as bestclfors anerure? ty t
Hay ce m on; my is pad; ss ace.
He bres w th fulithamed thethildiensprin nday shit'domo Pn.
I'mat we cudse berequgh n? w I his anlithe becy. fest at bi'st sharonindHivenc
e nind kee, me:
PLUENEE:
Be y epy choder blanchasetl m tir cureerdangas l
```

If you understand this, you basically understand the mechanism behind what a Transformer does. There's one small change left, and then we will build our first real Transformer!

## Self-Attention

The last step to understanding self attention is to *break symmetry*. Note that the way we currently have it, the `wei = x @ x.transpose(1, 2)` is symmetric, i.e. x matmul'd with itself is the same regardless of the way you order the `x`'s (as long as the dimensions make sense). In self-attention, instead of plainly using the token embeddings, we compute *projections* of `x` into `queries, keys,` and `values`. In self-attention, the queries attend to the keys to compute our weights for the weighted average. And instead of averaging the embeddings themselves, we average over the `values` to get our final contextualized embeddings.

Let's look at a small example

In [123…
```
q = nn.Linear(2, 2)
k = nn.Linear(2, 2)
v = nn.Linear(2, 2)

T, C = 4, 2
inp = torch.randn((T,C)).float() # shape: T, C
print(f'Input seq: {inp}')
```

```
Input seq: tensor([[-1.0394,  1.2911],
        [-1.7285, -0.2545],
        [-1.1180,  0.9540],
        [ 2.9836,  0.6682]])
```

In [123…
```
queries = q(inp)
keys = k(inp)
vals = v(inp)

print(f'Queries: {queries}') # shape: T, C
print(f'Keys: {keys}') # shape: T, C
print(f'Values: {vals}') # shape: T, C
```

```
Queries: tensor([[ 0.3488,  0.7321],
        [-0.0761, -0.0067],
        [ 0.2861,  0.5970],
        [ 1.9730,  1.9966]], grad_fn=<AddmmBackward0>)
Keys: tensor([[-0.9661, -0.4352],
        [-0.1405, -0.8073],
        [-0.7698, -0.4804],
        [ 0.3425,  1.5715]], grad_fn=<AddmmBackward0>)
Values: tensor([[ 0.6862,  1.2008],
        [ 0.1113,  1.0918],
        [ 0.5217,  1.1424],
        [-1.8952, -0.9225]], grad_fn=<AddmmBackward0>)
```

In [124…
```
wei = queries @ keys.T # shape: T, T
print(wei)
```

```
tensor([[-0.6556, -0.6401, -0.6202,  1.2700],
        [ 0.0764,  0.0161,  0.0618, -0.0366],
        [-0.5362, -0.5222, -0.5070,  1.0362],
        [-2.7749, -1.8891, -2.4779,  3.8135]], grad_fn=<MmBackward0>)
```

In [124…
```
tril = torch.tril(torch.ones(T, T))  # remove weights for elements past the cu
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1) # normalize weights to sum to 1
wei
```

Out[1241]:
```
tensor([[1.0000, 0.0000, 0.0000, 0.0000],
        [0.5151, 0.4849, 0.0000, 0.0000],
        [0.3285, 0.3332, 0.3383, 0.0000],
        [0.0014, 0.0033, 0.0018, 0.9935]], grad_fn=<SoftmaxBackward0>)
```

In [124…
```
wei @ vals # new embeddings (shape T, C)! we can project this or repeat the pr
```

Out[1242]:
```
tensor([[ 0.6862,  1.2008],
        [ 0.4074,  1.1480],
        [ 0.4390,  1.1447],
        [-1.8805, -0.9091]], grad_fn=<MmBackward0>)
```

## Practice: find the output embeddings (by hand) using self-attention for the following setup

$$T = 2, C = 2, X_{inp} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{TxC}$$

$$Q_{proj} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, K_{proj} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, V_{proj} = \begin{bmatrix} -1 & 1 \\ 0 & 3 \end{bmatrix}$$

$$\forall z \in \mathbb{R}^n, softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

## Answer:

$$Q = X_{inp}Q_{proj} = X_{inp} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

$$K = X_{inp}K_{proj} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

$$V = X_{inp}V_{proj} = \begin{bmatrix} -1 & 1 \\ -1 & 4 \end{bmatrix}$$

$$A = QK^T = \begin{bmatrix} 1 & 1 \\ 2 & 3 \end{bmatrix}$$

$$W = \text{softmax}(A \cdot \begin{bmatrix} 1 & -inf \\ 1 & 1 \end{bmatrix}) \begin{bmatrix} 1 & 0 \\ \frac{e^2}{e^2+e^3} & \frac{e^3}{e^2+e^3} \end{bmatrix}$$

$$X_{out} = WV = \begin{bmatrix} -1 & 1 \\ -1 & \frac{e^2+4e^3}{e^2+e^3} \end{bmatrix} \approx \begin{bmatrix} -1 & 1 \\ -1 & 3.912 \end{bmatrix}$$

In [114…

```
xi = torch.tensor([[1, 0], [1, 1]]).float()
qp = torch.eye(2).float()
kp = torch.eye(2).float()
kp[0,1] = 1
vp = torch.tensor([[-1, 1], [0, 3]]).float()

q = xi @ qp
k = xi @ kp
v = xi @ vp


print('q = ', q)
print('k = ', k)
print('v = ', v)
a = q @ k.T
print('a =', a)
w = a
w[0, 1] = float('-inf')
w = F.softmax(w, dim=-1)
w
print('w = ', w)
out = w @ v
print('out = ', out)
```

```
q =  tensor([[1., 0.],
        [1., 1.]])
k =  tensor([[1., 1.],
        [1., 2.]])
v =  tensor([[-1.,  1.],
        [-1.,  4.]])
a = tensor([[1., 1.],
        [2., 3.]])
w =  tensor([[1.0000, 0.0000],
        [0.2689, 0.7311]])
out =  tensor([[-1.0000,  1.0000],
        [-1.0000,  3.1932]])
```

## Implementing our Transformer

Let's use these lessons to implement a transformer with self-attention. One last thing to mention: we add `Feed Forward` layers after our self-attention, which is basically just an MLP/vanilla neural network. This allows the model to further make any modifications to the representations that aren't dependent on other tokens. Also, another important thing to note: we can repeat this `self attention` process. A neural network can have *multiple* self-atn + FFN blocks stacked on top of each other. This allows the model to learn very complex relationships between tokens. For reference, GPT-3 had 96 attention blocks stacked on top of each other.

We'll do 3 layers of attention + MLP in between each attention layer. The following implementation is courtesy of Andrej Karpathy

In [124…

```python
n_embd = 64
n_head = 1
n_layer = 3

@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

class Attention(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
```

```
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_s

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B,
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T,
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C)
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out

class FeedFoward(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, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
        )

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

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

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        self.sa = Attention(n_embd)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class SimpleLMWithAttention(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a l
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in ra
        self.lm_head = nn.Linear(n_embd, vocab_size)
```

```
    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        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, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # 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 sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

m = SimpleLMWithAttention()
```

In [124…

```
batch_size = 128

optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
for steps in range(10000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')
    # print(xb.shape)
    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
    if steps%500 == 0:
        print('step', steps, 'loss', loss.item())

print(loss.item())
```

```
step 0 loss 4.651000022888184
step 500 loss 2.0906076431274414
step 1000 loss 1.9890193939208984
step 1500 loss 1.9192143678665161
step 2000 loss 1.8416926860809326
step 2500 loss 1.782658338546753
step 3000 loss 1.7743024826049805
step 3500 loss 1.80558443069458
step 4000 loss 1.8363431692123413
step 4500 loss 1.789283037185669
step 5000 loss 1.8259267807006836
step 5500 loss 1.722832202911377
step 6000 loss 1.7234731912612915
step 6500 loss 1.7221698760986328
step 7000 loss 1.755784034729004
step 7500 loss 1.7279287576675415
step 8000 loss 1.702441692352295
step 8500 loss 1.7104332447052002
step 9000 loss 1.6752803325653076
step 9500 loss 1.6535974740982056
1.6962977647781372
```

In [124…
```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_t
```

```
GLOUCESTER:
No, eyess,
My lord, leaves, reasons foot a pare surment
here upon plain,
I we'll as 'tis that were whelp which thought lisand this then hears
And of chils shame told on than ill England, let's heart set thouse for queen,
'Capul bade.

DUKE Lady so be this petieung aluse cortain the bed: he tis age and her feel l
ain past of love quarry's bad it is a king-cursed:
What the dound his bawd,
To be oth Igent for most on pause, less from of uncle, are ussue!
But, ands thee? Thou knave forew.
```

In [124…
```
input_txt = "ALYSSA LIU: thine hath infected me with Covid!\n\nLORD CASEY: thou
print(input_txt)
ctx = encode(input_txt)
```

```
ALYSSA LIU: thine hath infected me with Covid!

LORD CASEY: thou art
```

In [124…
```
print(decode(m.generate(idx = torch.tensor(ctx).unsqueeze(0).long(), max_new_t
```

```
ALYSSA LIU: thine hath infected me with Covid!

LORD CASEY: thou art he more: there
Here.

KING RICHARD III:
How what he wife trovice parous head you, faints poes
Withose thee.
In me that would
Coments,
Widly, she oath is eur law great was was to God my omble she perfices: what's
in this groung'Thou to they cresolecess of a kingdom;
Be, eve fape the repering sun that baulty beg: as please thee liege,
Not mune shtience in more of head your garlet,
As with is cide buld I may thou shalt shall not, to could Richard oncome, comm
and lords. Come,
A partle Citiing with
```

In [121…

```python
batch_size = 128

optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
for steps in range(20000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')
    # print(xb.shape)
    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
    if steps%500 == 0:
        print('step', steps, 'loss', loss.item())

print(loss.item())
```

```
step 0 loss 1.6454037427902222
step 500 loss 1.6876657009124756
step 1000 loss 1.708350658416748
step 1500 loss 1.685848355293274
step 2000 loss 1.636243224143982
step 2500 loss 1.6772477626800537
step 3000 loss 1.6853232383728027
step 3500 loss 1.7382113933563232
step 4000 loss 1.6865171194076538
step 4500 loss 1.7063449621200562
step 5000 loss 1.7104920148849487
step 5500 loss 1.7120356559753418
step 6000 loss 1.6311984062194824
step 6500 loss 1.6379395723342896
step 7000 loss 1.7168763875961304
step 7500 loss 1.6899621486663818
step 8000 loss 1.681159496307373
step 8500 loss 1.7122695446014404
step 9000 loss 1.6832530498504639
step 9500 loss 1.6917122602462769
step 10000 loss 1.7628891468048096
step 10500 loss 1.6706066131591797
step 11000 loss 1.6901715993881226
step 11500 loss 1.705620288848877
step 12000 loss 1.6419752836227417
step 12500 loss 1.684890866279602
step 13000 loss 1.6462265253067017
step 13500 loss 1.6590042114257812
step 14000 loss 1.6221466064453125
step 14500 loss 1.6231399774551392
step 15000 loss 1.6877906322479248
step 15500 loss 1.753821849822998
step 16000 loss 1.66700279712677
step 16500 loss 1.6816389560699463
step 17000 loss 1.6505107879638672
step 17500 loss 1.5927612781524658
step 18000 loss 1.6304181814193726
step 18500 loss 1.728723168373108
step 19000 loss 1.6843644380569458
step 19500 loss 1.6999449729919434
1.6531990766525269
```

In [121…

```
input_txt = "ALYSSA LIU: thine hath infected me with Covid!\n\nLORD CASEY: thou
ctx = encode(input_txt)
print(decode(m.generate(idx = torch.tensor(ctx).unsqueeze(0).long(), max_new_t
```

```
ALYSSA LIU: thine hath infected me with Covid!

LORD CASEY: thou art that bled for grace pratorabused you, to the king, Warwic
k.
Lawfull glass.
I banished mine
In live Edward make what war
Done, not Roman.

MONTAGUE:
Ay, Anish his pardon, when rill-house. Thou show customb,
Here is deal here; 'twoman,
If why strokeuse your hand.
But thou look the cause.

EDWARD:
You we may more
Whom ensigal:
And held garden.

GLOUCESTER:
The from Franced his misted
As had corse.

KING RICHArt it?
Here whether action on us are am as oppother love
Villain,
Come, I am under well hourse: Hef what twent Green! Doth haughness do a, I say
that. I would achieves?' an old not straitors,
Lades the give does been, that your lust.

BUCKINGHAM:
My lord!
Is not hap give you grain is though the
life: I door:
A
moves, an ratelian's our tague go. In happy wolk one hold me save your as agai
nst thy was not with his sinued bless, as I be him, and DelphnishO, my enemy;
gaves and whith these she kill her majesty
And should the doubt,
Such sins,
By to husband death
Diotle range,
So the Caput
```

In [ ]: