Understanding the basic logic behind transformers and attention

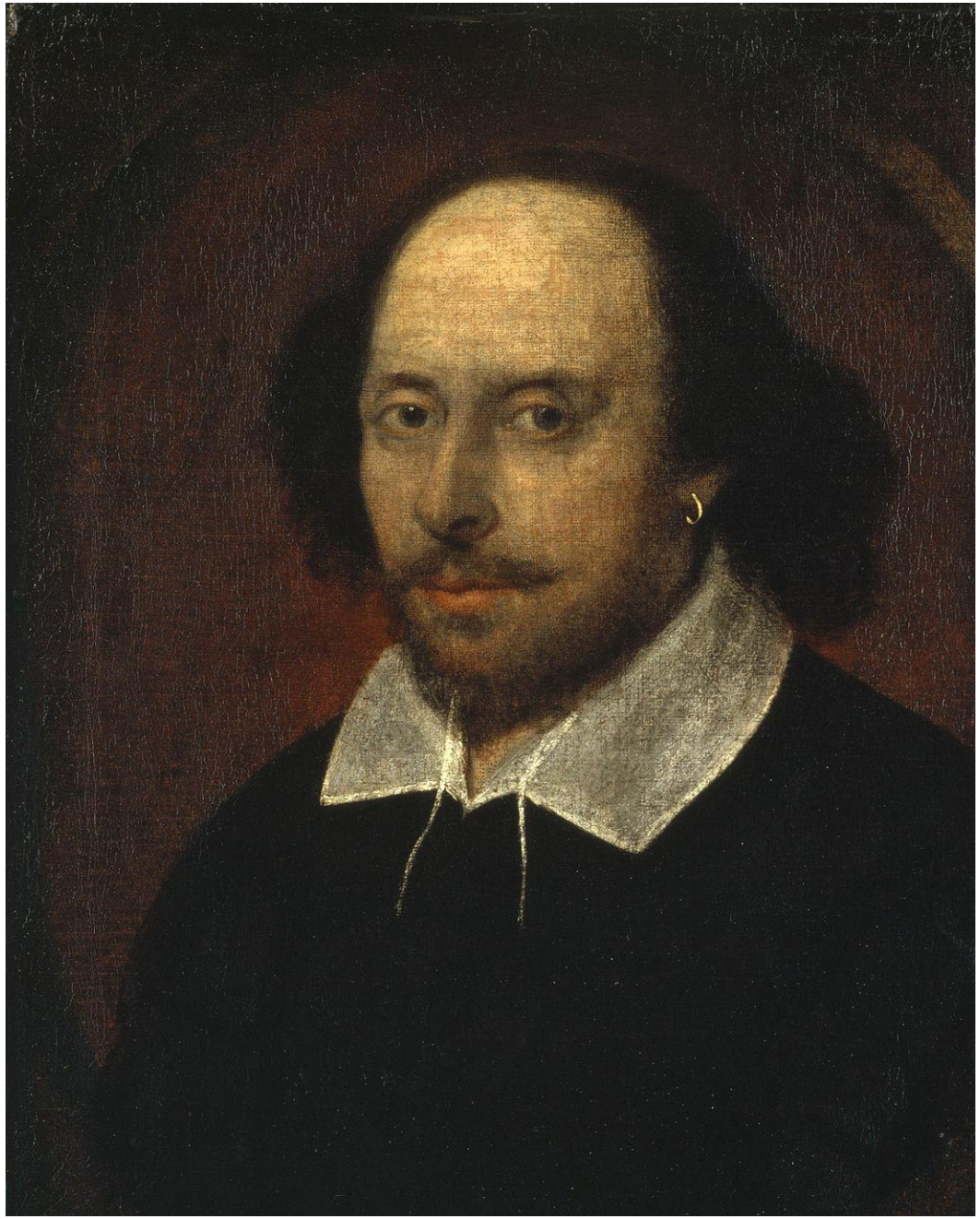
Tobias Höpfl

Video and sources used

- **Let's build GPT: from scratch, in code, spelled out.**
  https://www.youtube.com/watch?v=kCc8FmEb1nY
  By Andrej Karpathy (who works at OpenAI)
- Corresponding code by Karpathy on Colab:
  https://colab.research.google.com/drive/1JMLa53HDuA-i7ZBmqV7ZnA3c_fvtXnx-?usp=sharing
- Underlying paper:
  Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. In Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS 2017) (pp. 5998-6008).

Video by Karpathy

- Implementation of a small version of GPT from scratch
- Using PyTorch
- The model is trained only on the full work of Shakespeare
- It should produce text that looks Shakespeare-like



Encoder and decoder

- This is the „easy“ part we should already know from what we have done on NLP so far
- Here our vocabulary are characters, not words/subwords
- This is our vocabulary (65 characters):

```
!$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

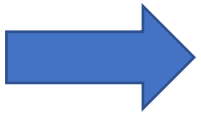
- Our „encoder“ and „decoder“ mapping from words to integers and back:

```
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder
decode = lambda l: ''.join([itos[i] for i in l])
```

- In addition we also should encode the position (not shown here)

Encoding the whole dataset

```
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
but they think we are too dear: the leanness that
afflicts us, the object of our misery, is as an
```



```python
# let's now encode the entire text dataset and store it into a torch.Tensor
import torch # we use PyTorch: https://pytorch.org
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
print(data[:1000]) # the 1000 characters we looked at earier will to the GPT look
```

```
torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59,  1, 39, 56, 43,  1, 39, 50, 50,
         1, 56, 43, 57, 53, 50, 60, 43, 42,  1, 56, 39, 58, 46, 43, 56,  1, 58,
        53,  1, 42, 47, 43,  1, 58, 46, 39, 52,  1, 58, 53,  1, 44, 39, 51, 47,
        57, 46, 12,  0,  0, 13, 50, 50, 10,  0, 30, 43, 57, 53, 50, 60, 43, 42,
         8,  1, 56, 43, 57, 53, 50, 60, 43, 42,  8,  0,  0, 18, 47, 56, 57, 58,
         1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 18, 47, 56, 57, 58,  6,  1, 63,
        53, 59,  1, 49, 52, 53, 61,  1, 15, 39, 47, 59, 57,  1, 25, 39, 56, 41,
        47, 59, 57,  1, 47, 57,  1, 41, 46, 47, 43, 44,  1, 43, 52, 43, 51, 63,
         1, 58, 53,  1, 58, 46, 43,  1, 54, 43, 53, 54, 50, 43,  8,  0,  0, 13,
        50, 50, 10,  0, 35, 43,  1, 49, 52, 53, 61,  5, 58,  6,  1, 61, 43,  1,
        49, 52, 53, 61,  5, 58,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47, 58,
```

Block size

- We define our block size to be 8
- This means that when we predict the next character, we can look back at maximum 8 characters (maximum context of 8 characters)
- We use each context as a separate training example (i.e. 1, 2…, 8 characters back)

```
[ ]  block_size = 8
     train_data[:block_size+1]
```

```
     tensor([18, 47, 56, 57, 58,  1, 15, 47, 58])
```

```
[ ]  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}")
```

```
     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
```

Language model

- We use a simple language model called the Bigram model
- It contains an embedding table of vocab_size * vocab_size
- So for each vocabulary (here: character) it should determine, what the probability is for each vocabulary to occur next
- E.g. for consonants it will be more probable to be followed by a vowel

```
class BigramLanguageModel(nn.Module):

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

Forward propagation

- We go to the row corresponding to the index in the embedding table
- The content is of dimension:

  batch_size *
  time (= block size) *
  channels (= size of vocabulary)

```
def forward(self, idx, targets=None):

    # idx and targets are both (B,T) tensor of integers
    logits = 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
```

Forward propagation

- We use cross entropy (negative log likelihood) as a loss function:

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$$

- Measures quality of prediction

```python
def forward(self, idx, targets=None):

    # idx and targets are both (B,T) tensor of integers
    logits = 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
```

Text generation

- For text generation we apply softmax to the logit and sample from the resulting distribution to determine the output (Softmax: exponentiate every element and divide by the sum):

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

- So there is a random element at this point which is why ChatGPT generates different output even if the input is the same

```
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
```

Generation

- The model can already generate text, which is not meaningful yet of course
- We can then also train it only based on the one preceding character, which is still not yet enough

```
oTo.JUZ!!zqe!
xBP qbs$Gy'AcOmrLwwt
p$x;Seh-onQbfM?OjKbn'NwUAW -Np3fkz$FVwAUEa-wzWC -wQo-R!v -Mj?,SPiTyZ;
rT'Fd,SBMZyOslg!NXeF$sBe,juUzLq?w-wzP-h
ERjjxlgJzPbHxf$ q,q,KCDCU fqBOQT
SV&CW:xSVwZv'DG'NSPypDhKStKzC -$hslxIVzoivnp ,ethA:NCCGoi
tN!ljjP3fwJMwNelgUzzPGJlgihJ!d?q.d
pSPYgCuCJrIFtb
jQXg
pA.P LP,SPJi
DBcuBM:CixjJ$Jzkq,OLf3KLQLMGph$O 3DfiPHnXKuHMlyjxEiyZib3FaHV-oJa!zoc'
D.?
```

Weighted aggregation

- We can take into account the last 8 (which is our block size) tokens
- We can take the average by doing a for loop or more efficiently by multiplying with a triangular matrix (upper right filled with 0s)
- E.g. the first and second column of the first row are empty, because we don‘t want to look at future tokens

```
# version 3: use Softmax
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)
xbow3 = wei @ x
torch.allclose(xbow, xbow3)
```

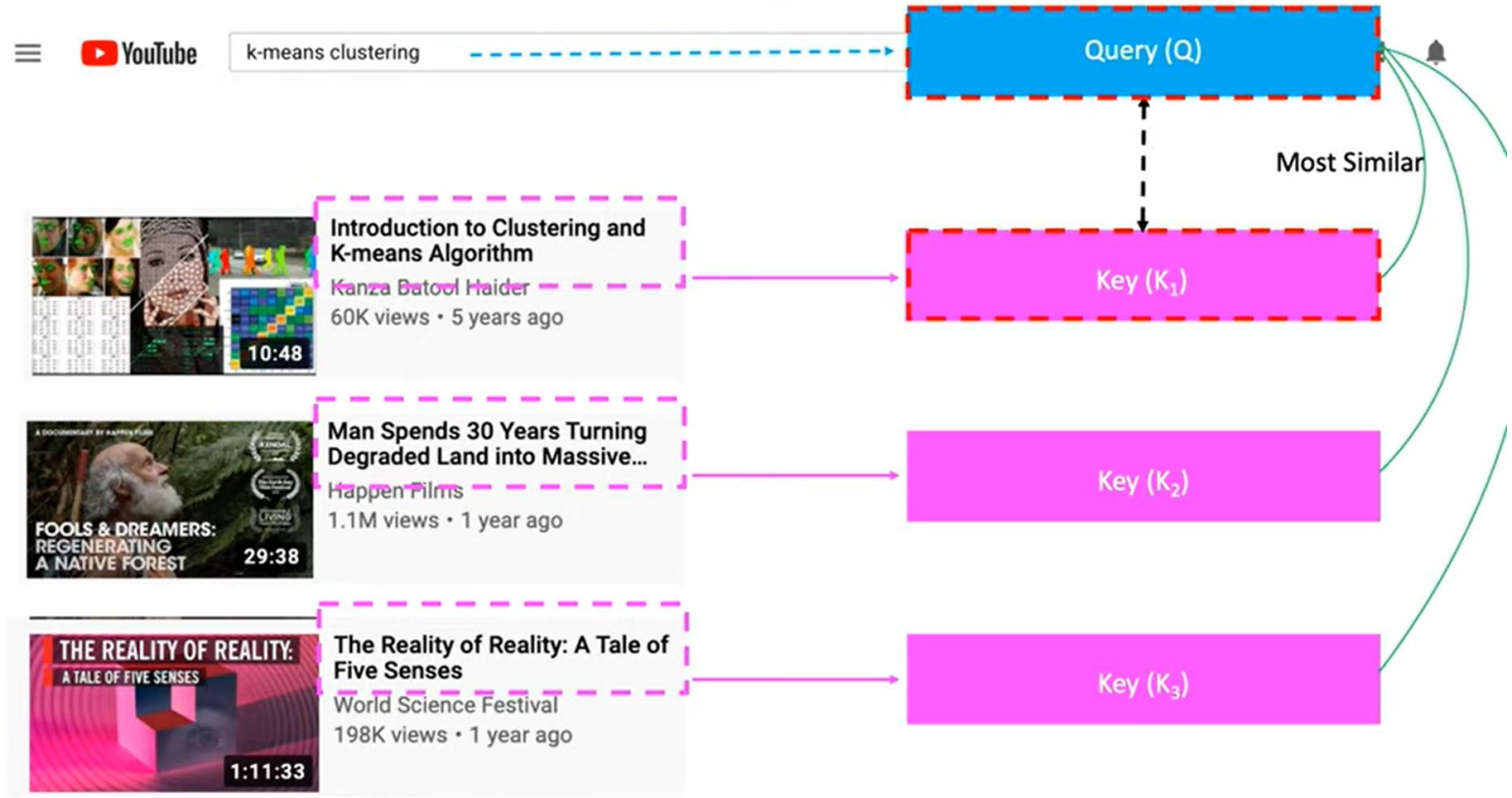
```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
```

Attention

- Equal aggregation is not the smartest way though
- In text not every letter or vocabulary has the same level of interest for another
- Some are more important for the contextual understanding than others
- Our tokens should instead should pay „attention“ more to specific other tokens in a data dependent way
- The resulting matrix (attention matrix) should look accordingly
- Attention is a communication mechanism

Self-attention: query and key

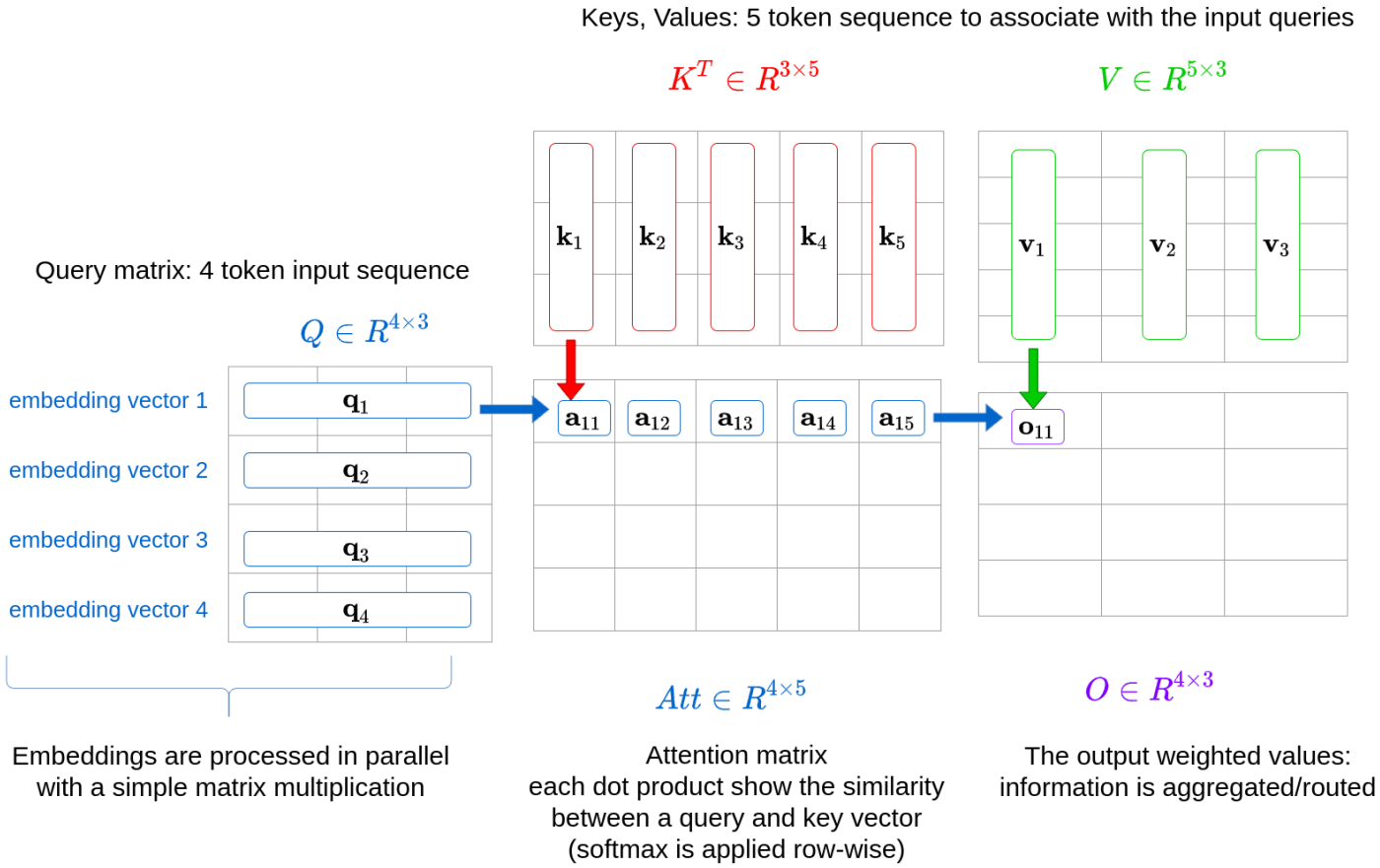
- Every token emits two vectors: a query and a key (+ a value)
- Query: „What kind of information am I interested in?“
- Key: „What do I contain?“
- Compare information retrieval (e.g. Youtube)



https://www.youtube.com/watch?v=mMa2PmYJlCo

Self-attention: query, key and value

- How to find the affinity
- Create dot product between query and each possible key
- Where the result is highest most attention should be put towards
- Value is the „private information“ that is multiplied to create the output



https://theaisummer.com/static/e497f0d469418119f9db9c53b9851e61/b9460/self-attention-explained.png

Self-attention

- Key, query and value are linear modules
- We then calculate the dot product between query and key
- `wei = wei.masked_fill(tril == 0, float('-inf'))` masks again the left upper half of the matrix (the future is unknown -> Decoder block, because we want to generate text)
- Self-attention because the same source (here x) produces query, key and value

```
# version 4: self-attention!
torch.manual_seed(1337)
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)
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
```

Multi-head attention

- Calculate multiple attentions at the same time and concatenate results

```python
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

Feed Forward

- Computation part that is done for every token at the end

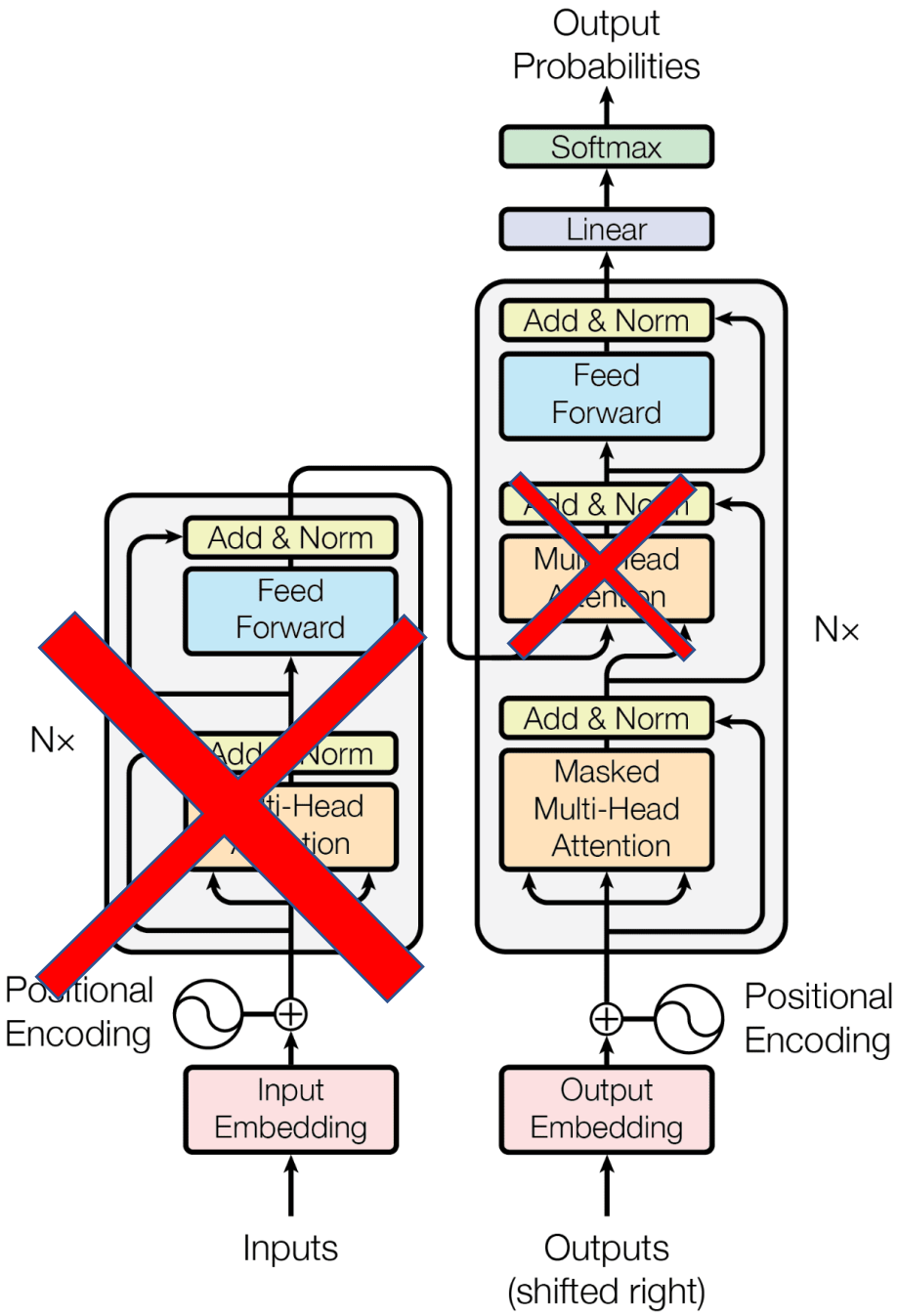
```
class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    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),
            nn.Dropout(dropout),
        )

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

Result

- Now all the basic code parts are ready that allow to build a transformer
- Although the final code is still a bit larger and more complex
- The final model can indeed produce Shakespeare-like text
- So far only text completor
- ChatGPT in addition had to be fine-tuned to be an assistant (-> reward model + multiple other training stages)

Wrap-up and high level overview

- On the left we see encoder, on the right the decoder
- For GPT only the decoder part is used (encoder-decoder is used e.g. for translation software)
- Input embeddings + positional encodings
- Multi-Head Attention
- Feed Forward Layer
- Finally: Linear and Softmax



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. In Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS 2017) (pp. 5998-6008).