A Small Language Model with Self Attention

PyTorch Implementation

Dr. Uzair Ahmad

Expected Outcomes

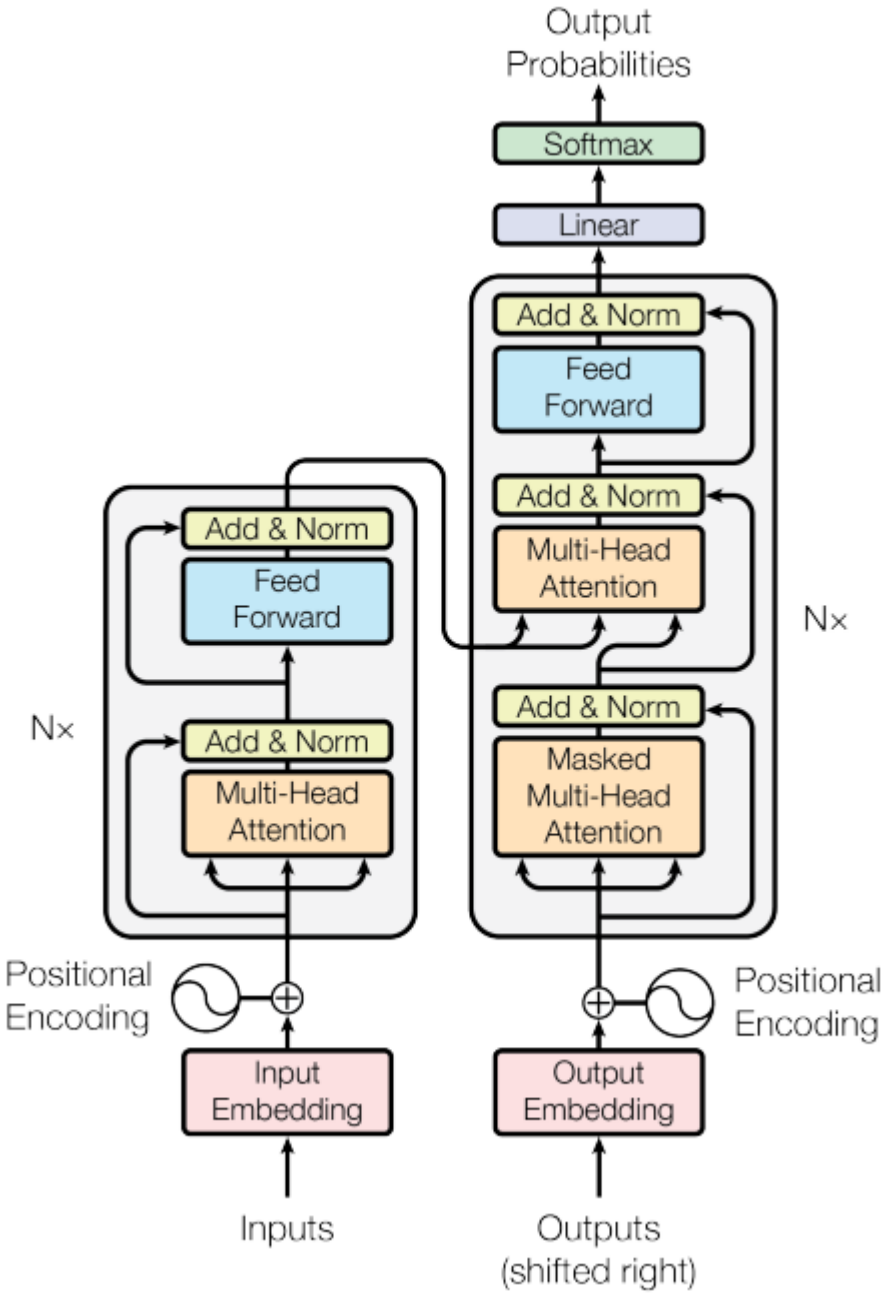
- Understand
  - the Transformer architecture
  - Self-attention mechanism
    - Concept and implementation
- Train & evaluate a “small” language model
  - Using considerable smaller inputs than vocabulary
- PyTorch Implementation

The Transformer

Neural Information Processing Systems
https://papers.neurips.cc › paper › 7181-attentio... PDF

Attention is All you Need - NIPS papers

by A Vaswani · Cited by 67472 — We propose a **new simple network architecture, the Transformer**, based solely on attention mechanisms, dispensing with recurrence and…

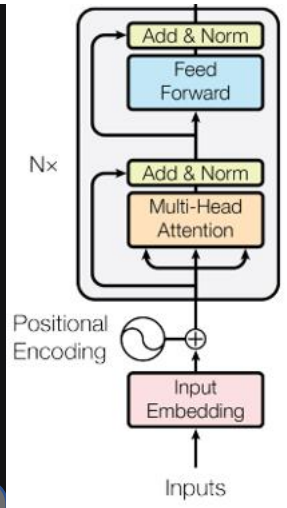


Prep 1

```
def prep(self, corpus):                                                    ⚠ 5  ✓ 7
    self.vocab = sorted(list(set(corpus)))
    self.vocab_size = len(self.vocab)
    c2i = {c: i for i, c in
           enumerate(self.vocab)}  # char c to integer i map. assign value i for every word
    i2c = {i: c for c, i in c2i.items()}  # integer i to char c map

    self.encoder = lambda doc: [c2i[c] for c in doc]
    self.decoder = lambda nums: ''.join([i2c[i] for i in nums])

    n = len(text)
    self.train_text = text[:int(n * 0.9)]
    self.val_text = text[int(n * 0.9):]

    self.train_data = torch.tensor(self.encoder(self.train_text), dtype=torch.long)
    self.val_data = torch.tensor(self.encoder(self.val_text), dtype=torch.long)
```

Prep 2

```
# look-up table for embeddings (vocab_size x embed_size)
# it will be mapping each token id to a vector of embed_size
# a wrapper to store vector representations of each token
self.token_embeddings_table = \
    nn.Embedding(self.vocab_size, self.embed_size)
# self-attention head
self.sa_head = self.SelfAttentionHead(in_size=self.embed_size,
                                      out_size=self.sa_head_size)
# linear projection of sa_head output to vocabulary
self.linear_sahead_to_vocab = nn.Linear(self.sa_head_size, self.vocab_size)
```



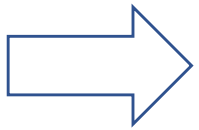
Visualizing Embedding matrix

		a	b	c	d	e	f	g	h	i	j
a	1	0.23	0.16	0.11	0.09	0	0.07	0.02	0.02	0.09	0.2
b	2	0.18	0.23	0.16	0.14	0.18	0.14	0.07	0.16	0.02	0.14
c	3	0.14	0.18	0.23	0.23	0.23	0.07	0	0.23	0.05	0.14
d	4	0.23	0.14	0.07	0.23	0	0.23	0.02	0.16	0.02	0.18
e	5	0.05	0.18	0.14	0.11	0.23	0.18	0.11	0.16	0.16	0.16
f	6	0.11	0.16	0.02	0.11	0.16	0.23	0.05	0	0.2	0.14
g	7	0.02	0.2	0.16	0.09	0.07	0.07	0.23	0.07	0.16	0.05
h	8	0	0.07	0.16	0.07	0	0.07	0.09	0.23	0.05	0.11
i	9	0.02	0.05	0.18	0.14	0.05	0.07	0	0.23	0.23	0.11
j	10	0.05	0.11	0.16	0.23	0.11	0.07	0.2	0.14	0.18	0.23

c2i map

10 x 10

|V|x**|V|** Embeddings



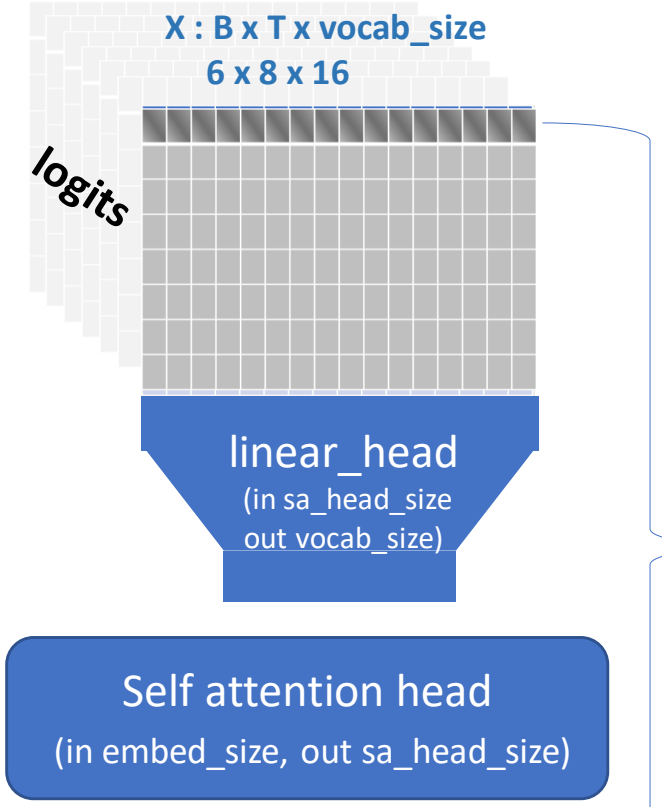
		d1	d2	d3	d4	dn
a	1	-0.5	-1	0.82	-0.3	1.62
b	2	0.71	0.8	0.85	0	-0.6
c	3	0	-1.4	0.56	-0.3	-0.2
d	4	0.96	-0.4	-1.8	-0.8	1.66
e	5	0	-0.5	0	-1.6	-0.8
f	6	-0.7	-0.8	-1.4	1.27	1.2
g	7	0.78	0	-0.4	-0	-0.2
h	8	1.43	-0.1	-0.7	-0.4	1
i	9	-1.3	1.68	0.34	0.1	-1.3
j	10	0.99	0.03	1.79	0	-1.3

c2i map

|V|x**n** Embeddings

Computation

Communication



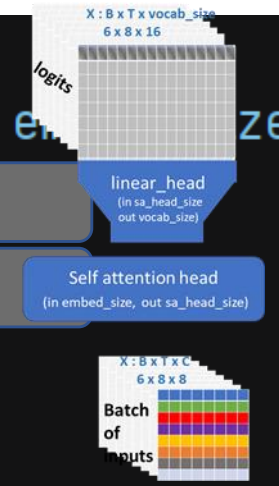
Which token comes next (most likely) if the current token is input

Fit the model

```
def fit(self, train_iters=100, eval_iters=10, lr=0.0001):
    """..."""
    optimizer = torch.optim.Adam(self.parameters(), lr=lr)
    for iteration in range(train_iters):
        if iteration % eval_iters == 0:
            avg_loss = self.eval_loss(eval_iters)
            print(f"iter {iteration}: train {avg_loss['train']} val {avg_loss['eval']}")
        inputs, targets = self.get_batch(split='train')
        _, ce_loss = self(inputs, targets)
        optimizer.zero_grad(set_to_none=True)  # clear gradients of previous step
        ce_loss.backward()  # propagate loss back to each unit in the network
        optimizer.step()  # update network parameters w.r.t the loss
```

Forward pass

```
    def forward(self, in_ids, target=None):
        in_ids_emb = self.token_embeddings_table(in_ids)  # batch_size x e
        sa_head_outputs = self.sa_head(in_ids_emb)  # communicate
        logits = self.linear_sahead_to_vocab(sa_head_outputs)  # compute
        if target is None:
            ce_loss = None
        else:
            batch_size, input_length, vocab_size = logits.shape
            logits_ = logits.view(batch_size * input_length, vocab_size)
            targets = target.view(batch_size * input_length)
            ce_loss = F.cross_entropy(logits_, targets)
        return logits, ce_loss
```

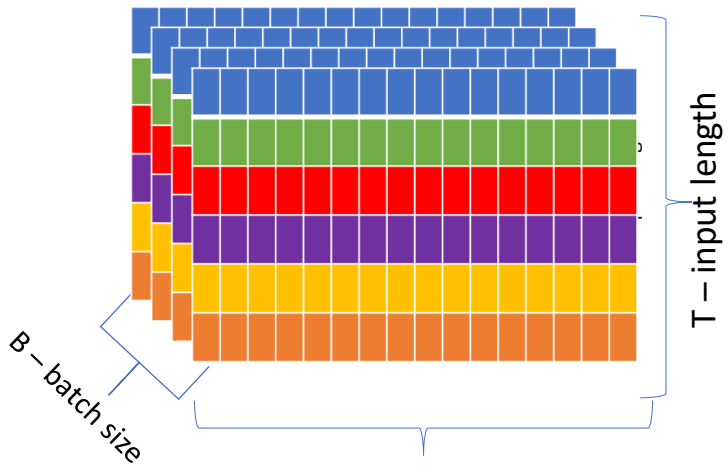


Evaluation loss

```python
@torch.no_grad()  # tell torch not to prepare for back-propagation
def eval_loss(self):
    perf = {}
    # set dropout and batch normalization layers to evaluation mode before running inf
    self.eval()
    for split in ['train', 'eval']:
        losses = torch.zeros(self.eval_iters)
        for k in range(self.eval_iters):
            tokens, targets = self.get_batch(split)  # get random batch of inputs and
            _, ce_loss = self(tokens, targets)  # forward pass
            losses[k] = ce_loss.item()  # the value of loss tensor as a standard Pytho
        perf[split] = losses.mean()
    self.train()  # turn-on training mode-
    return perf
```

logits.view()

token_rep

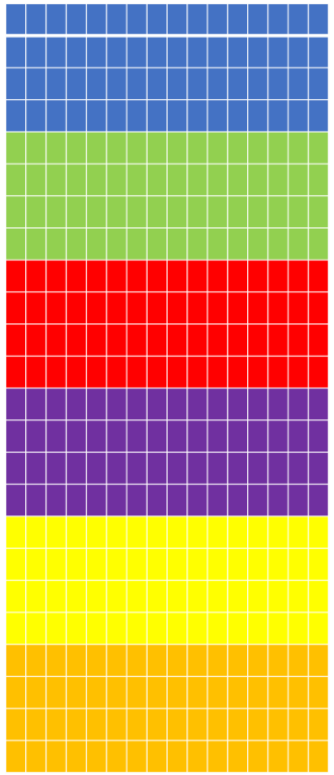


C – vocab_size

B x T x C

4 x 6 x 16

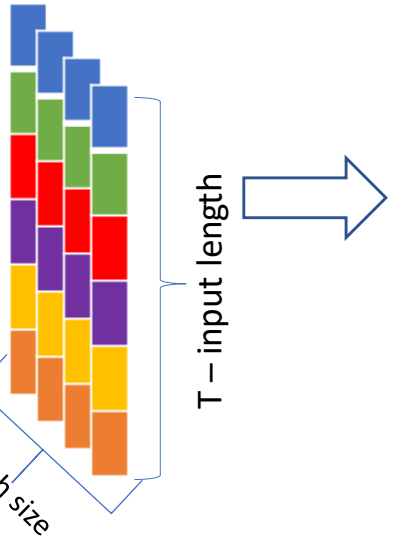
token_rep



BT x C

24 x 16

target



B x T

4 x 6

target

token_rep = token_rep.view(batch_size * input_length, vocab_size)

targets = target.view(batch_size * input_length)

Cross Entropy

$$\sum_{c=1}^{C} p(y_c).\log(p(\hat{y}_c))$$

- Logits
- Softmax
- Negative Log Likelihood

```
logits = torch.tensor([[0.1, 0.5, 0.2, .1]])
targets = torch.tensor([1])
F.cross_entropy(logits, targets)
```

```
tensor(1.1254)
```

```
import numpy as np
nll = 0
target = 1

den = 0
for logit_c in logits[0].tolist():
    den += np.exp(logit_c)
for c, logit_c in enumerate(logits[0].tolist()):
    if c == target:
        P_c = np.exp(logit_c)/den # softmax/probability of Yc
        nll = - np.log(P_c)
print(nll)
```

```
1.1254029644614048
```

Self attention head – K, Q, V

```python
class SelfAttentionHead(nn.Module):
    def __init__(self, in_size, out_size):
        """..."""
        super().__init__()
        self.head_size = out_size
        self.K = nn.Linear(in_size, self.head_size, bias=False)
        self.Q = nn.Linear(in_size, self.head_size, bias=False)
        self.V = nn.Linear(in_size, self.head_size, bias=False)

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

Self attention head – Forward pass

```
class SelfAttentionHead(nn.Module):
    def __init__(self, in_size, out_size):...

    def forward(self, x):
        keys = self.K(x)
        queries = self.Q(x)
        ...
        keys_t = keys.transpose(1, 2)
        autocorrs = (queries @ keys_t) * (self.head_size ** -0.5)  # (batch_size x input_length
        '''...'''
        autocorrs = torch.tril(autocorrs)
        autocorrs = autocorrs.masked_fill(autocorrs == 0, float('-inf'))
        autocorrs = torch.softmax(autocorrs, dim=-1)
        values = self.V(x)  # (batch_size x input_length x head_size)
        out = autocorrs @ values
        return out
```

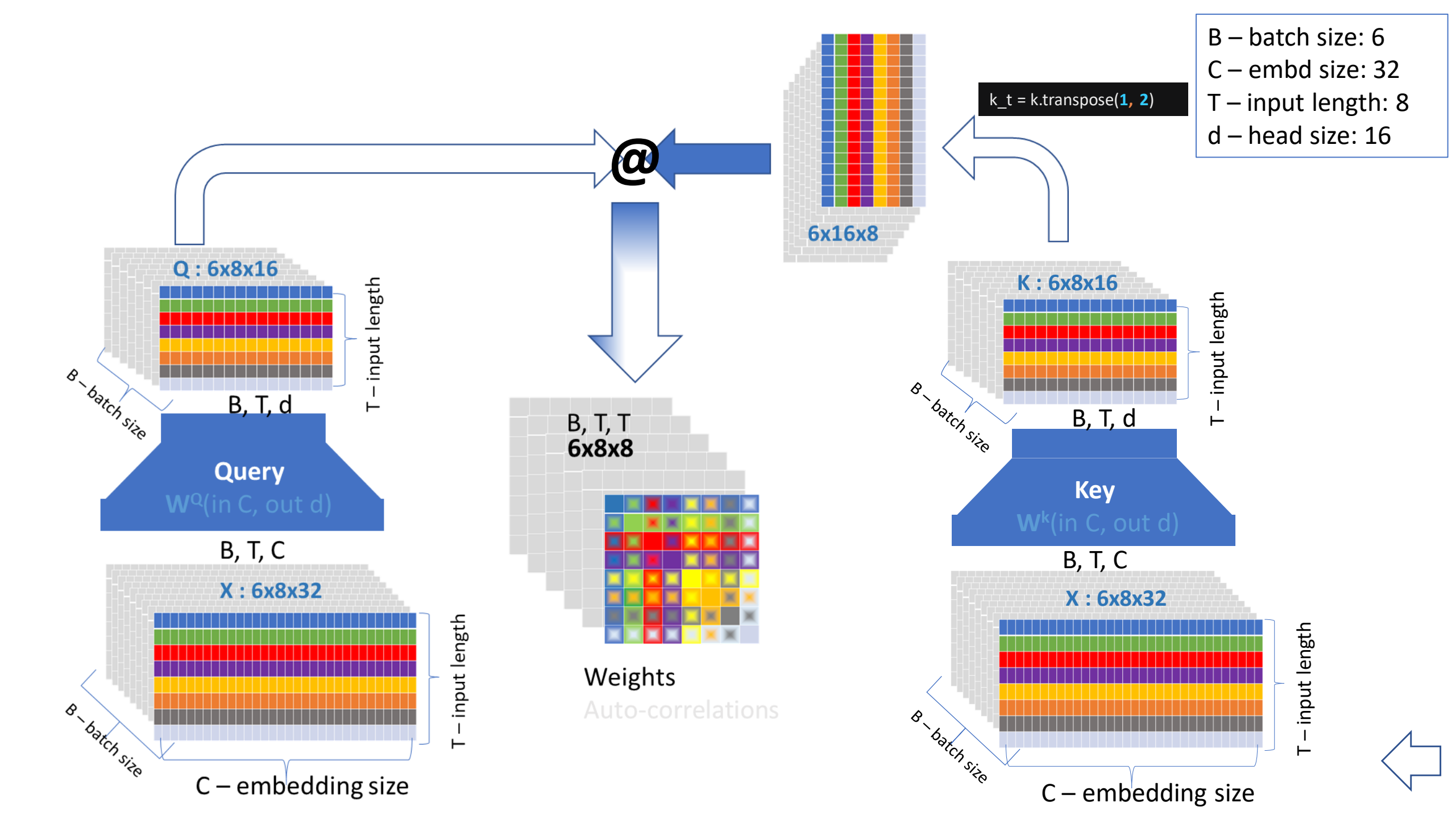
Scaling the attention by head_size$^{1/2}$

- Scaled attention:
  - Prevent peaky inputs to SoftMax . Why ?

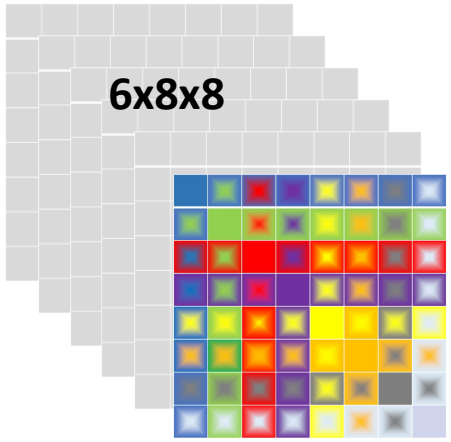
```
1   import torch
2   logits = torch.tensor([0.1,0.4,-0.9, 0.02, 0.35, -0.62])
3   print(torch.softmax(logits, dim=0).numpy().round(2)) # diffused
4   print(torch.softmax(logits*100, dim=0).numpy().round(2)) # One-hot-encoding
```

```
[0.18 0.25 0.07 0.17 0.24 0.09]
[0.   0.99 0.   0.   0.01 0.  ]
```

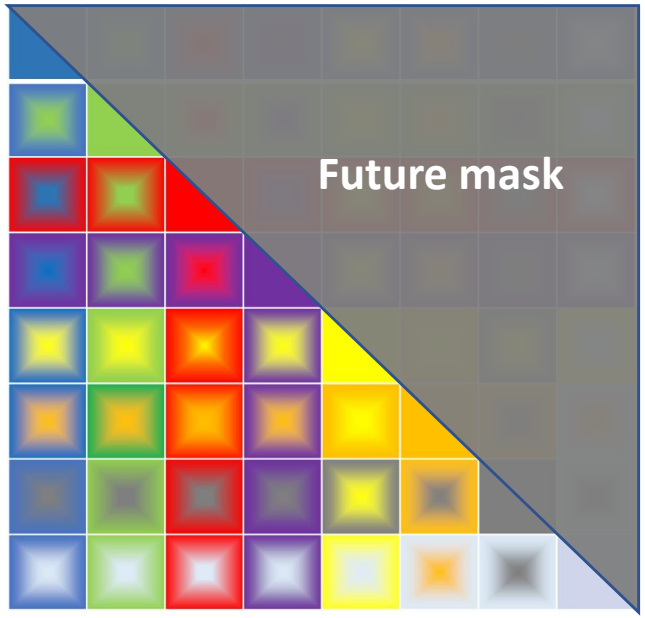
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

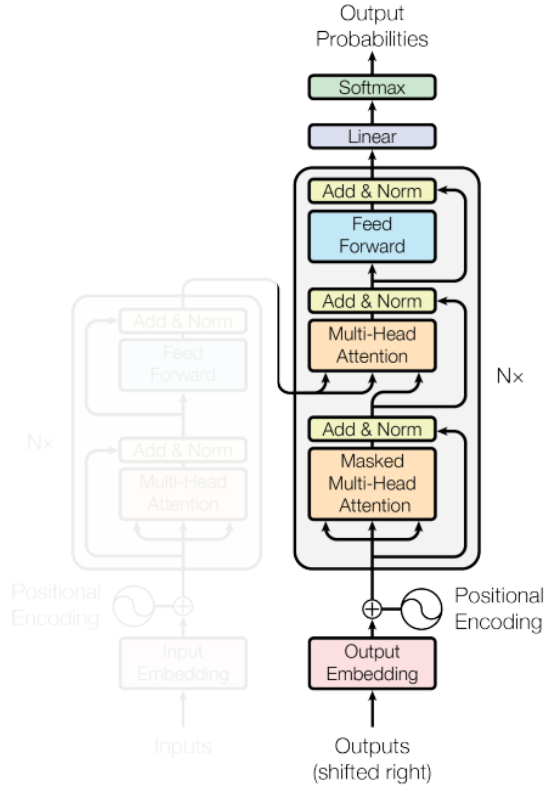

B – batch size: 6
C – embd size: 32
T – input length: 8
d – head size: 16
k_t = k.transpose(1, 2)
@
6x16x8
Q : 6x8x16
K : 6x8x16
B, T, d
T – input length
B – batch size
Query
W^Q(in C, out d)
Key
W^k(in C, out d)
B, T, C
X : 6x8x32
C – embedding size
B, T, T
6x8x8
Weights
Auto-correlations

Future mask



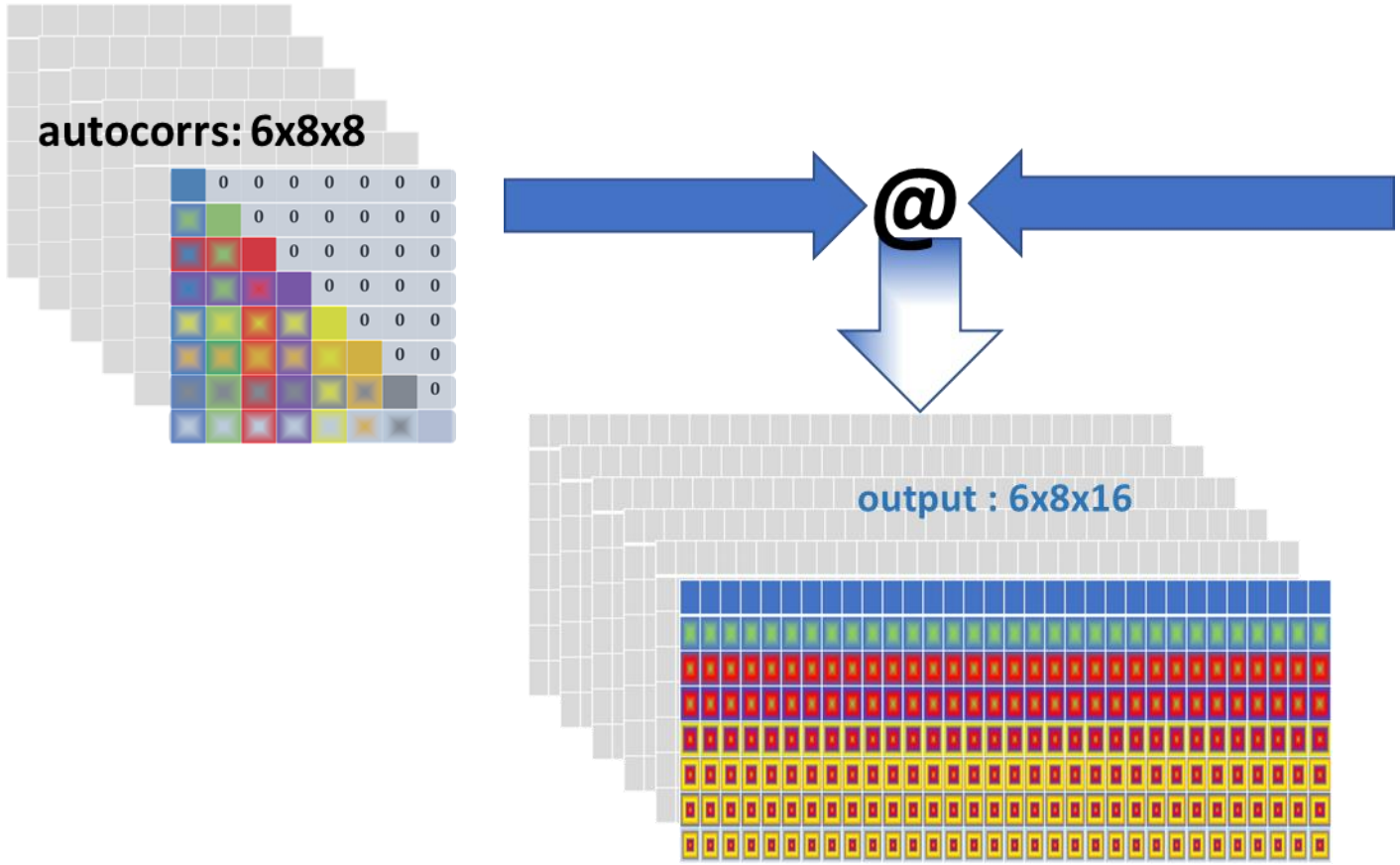
Weights
Auto-correlations






```
autocorrs = torch.tril(autocorrs)
autocorrs = autocorrs.masked_fill(autocorrs == 0, float('-inf'))
autocorrs = torch.softmax(autocorrs, dim=-1)
```

B – batch size: 6
C – emb size: 32
T – input length: 8
d – head size: 16



Comparison

```
iter 0 train 3.752662181854248 val 3.8865482807159424
iter 1000 train 0.6874994039535522 val 0.7462117075920105
iter 2000 train 0.683384358882904 val 0.7035092711448669
iter 3000 train 0.6830695867538452 val 0.7749010324478149
iter 4000 train 0.6907606720924377 val 0.6953048706054688

 the the brof la dofofown la la lazy qui
```

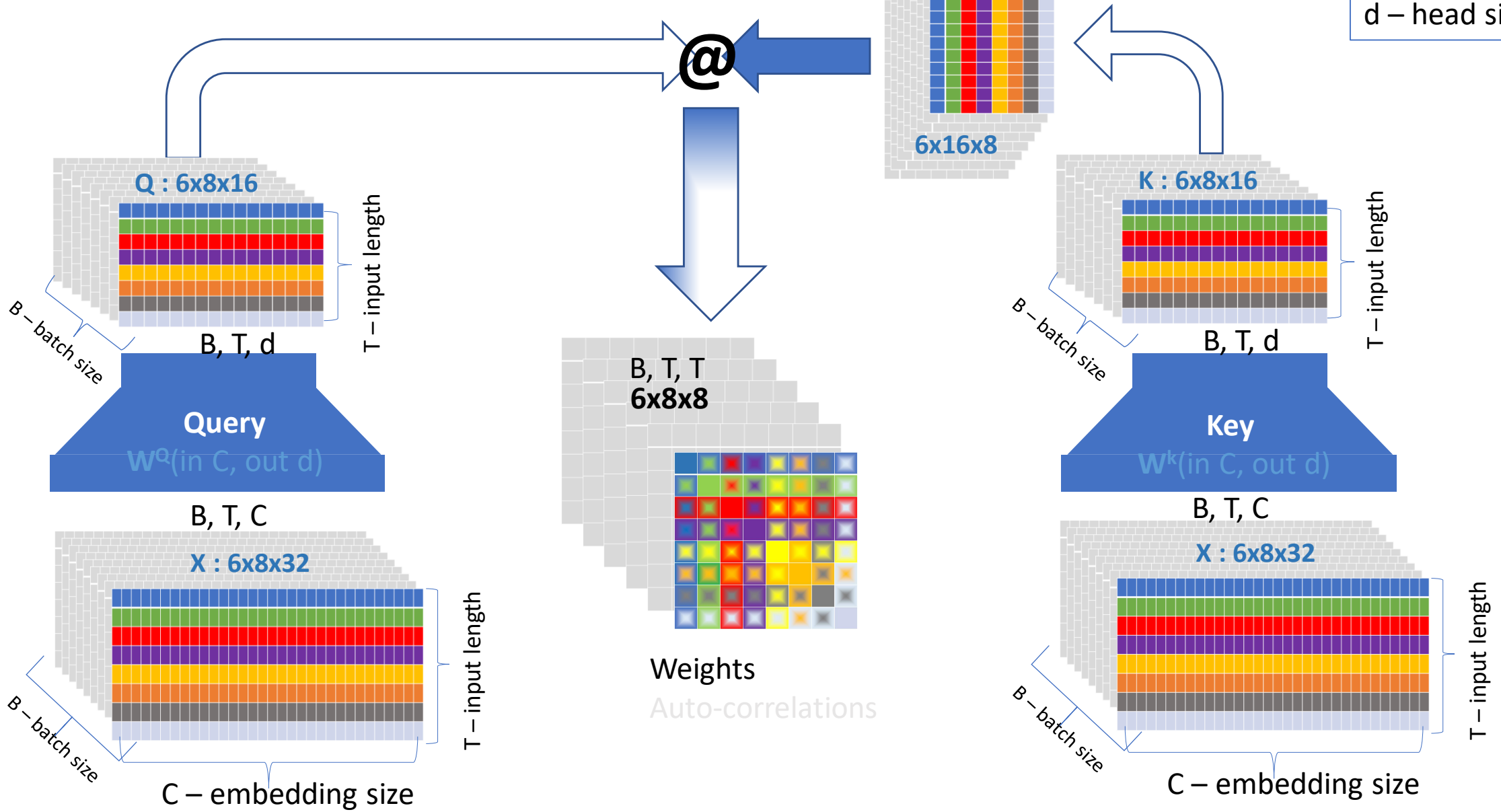
BiGram: vocab_size x vocab_size

```
iter 0: train 3.343690872192383 val 3.369588851928711
iter 1000: train 0.19767841696739197 val 0.19416601955890656
iter 2000: train 0.15668639540672302 val 0.17578212916851044
iter 3000: train 0.13835833966732025 val 0.12090074270963669
iter 4000: train 0.13430319726467133 val 0.1458485871553421

a quick brown brown fown fox fox juick f
```

SelfAttention: vocab_size x embed_size

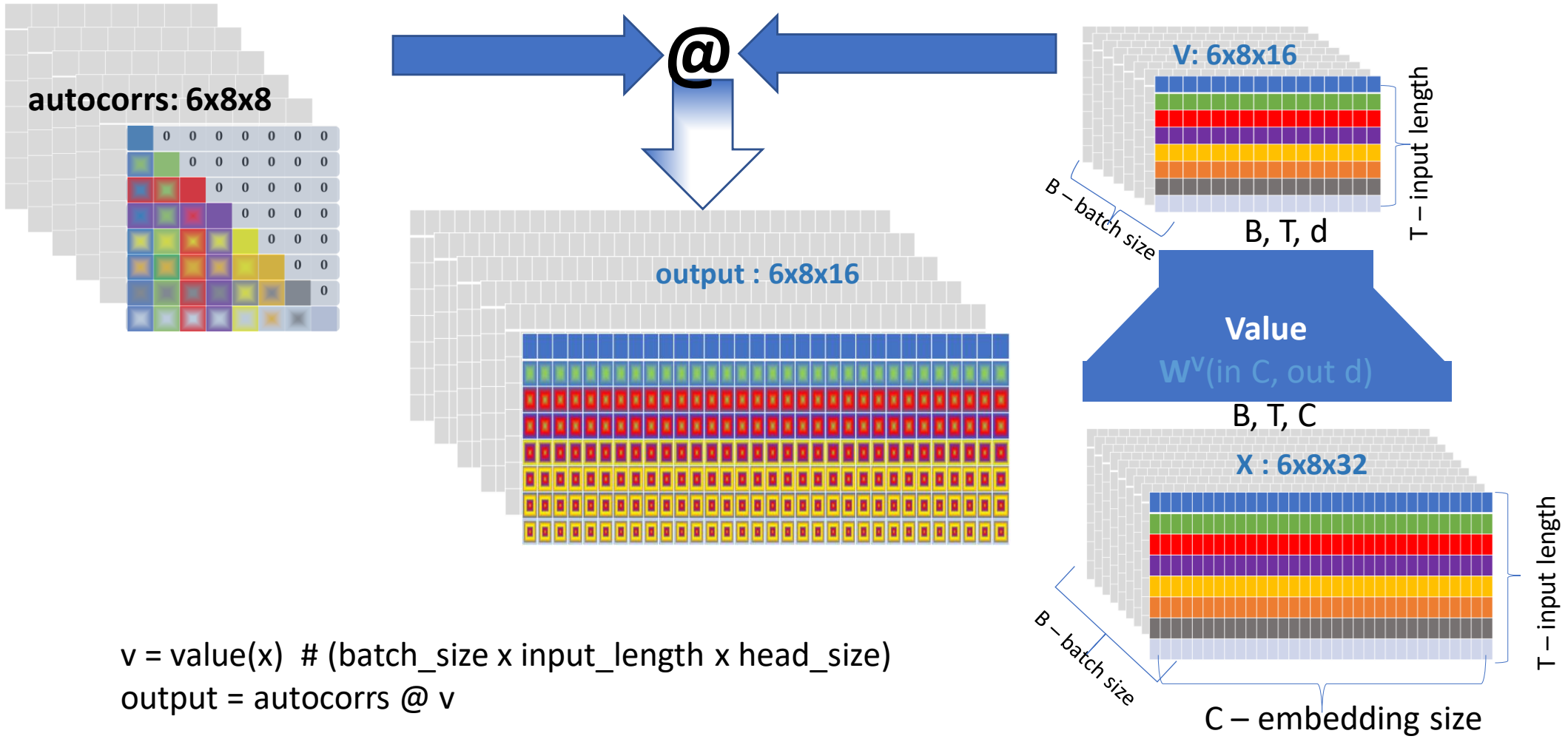
Query @ Key

B – batch size: 6
C – embd size: 32
T – input length: 8
d – head size: 16



autocorrs @ value

B – batch size: 6
C – emb size: 32
T – input length: 8
d – head size: 16



v = value(x) # (batch_size x input_length x head_size)
output = autocorrs @ v