Bigram Language Model

PyTorch Implementation

Dr. Uzair Ahmad

Run

```
model = BigramLanguageModel()
model = model.to(model.device)
model.prep(text)
input_batch, output_batch = model.get_batch(split='train')
_, _ = model(input_batch, output_batch)
model.fit()
outputs = model.generate(context_tokens=torch.zeros((1, 1), dtype=torch.long,
                         device=model.device), max_new_tokens=100)
```

Character to Integer

```
vocab = sorted(list(set(text)))
self.vocab_size = len(vocab)
# char c to integer i map. assign value i for every word in vocab
ctoi = {c: i for i, c in enumerate(vocab)}
# integer i to char c map
itoc = {i: c for c, i in ctoi.items()}
```

- Tiktoken
- SentencePiece

Preparing the text

```
def prep(self, text):
    vocab = sorted(list(set(text)))
    self.vocab_size = len(vocab)
    ctoi = {c: i for i, c in enumerate(vocab)} # char c to integer i map. assign value
    itoc = {i: c for c, i in ctoi.items()} # integer i to char c map

    self.encoder = lambda text: [ctoi[c] for c in text]
    self.decoder = lambda nums: ''.join([itoc[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)

    # look-up table for embeddings (vocab_size x vocab_size)
    # the model will turning each input token into a vector of size vocab_size
    # a wrapper to store vector representations of each token
    self.token_embeddings_table = nn.Embedding(self.vocab_size, self.vocab_size)
```

Why not shuffling ?

Massive tensors
Why validation ?

Visualizing Embedding Matrix

```
1   import torch
2   from torch import nn
3
4   embedding = nn.Embedding(5, 5)
5   print(embedding.weight)
```

```
Parameter containing:
tensor([[-0.3577,  1.5810, -0.4393,  0.2227, -0.6662],
        [-0.4664, -1.0360, -0.4156,  0.1769,  0.8236],
        [-0.4090, -0.6741, -2.8593,  0.8315,  2.4842],
        [ 0.5647,  1.0296, -0.0361,  1.4763, -0.0302],
        [-0.4024,  0.1250, -0.9818,  0.7519, -1.4089]], requires_grad=True)
```

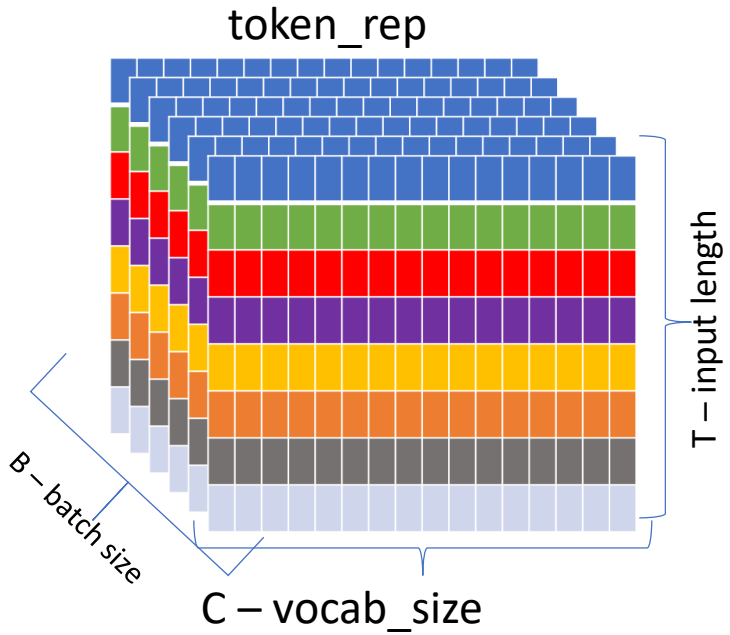
```
1   print(embedding(torch.tensor([1])))
```

```
tensor([[-0.4664, -1.0360, -0.4156,  0.1769,  0.8236]],
       grad_fn=<EmbeddingBackward0>)
```

Batch-ify the input

```
def get_batch(self, split='train'):
    data = self.train_data if split == 'train' else self.val_data
    # get random chunks of length batch_size from data
    ix = torch.randint(len(data) - self.input_length,
                       (self.batch_size,))
    inputs_batch = torch.stack([data[i:i + self.input_length] for i in ix])
    targets_batch = torch.stack([data[i + 1:i + self.input_length + 1] for i in ix])
    inputs_batch = inputs_batch.to(self.device)
    targets_batch = targets_batch.to(self.device)
    return inputs_batch, targets_batch
```

Visualizing the **batch**

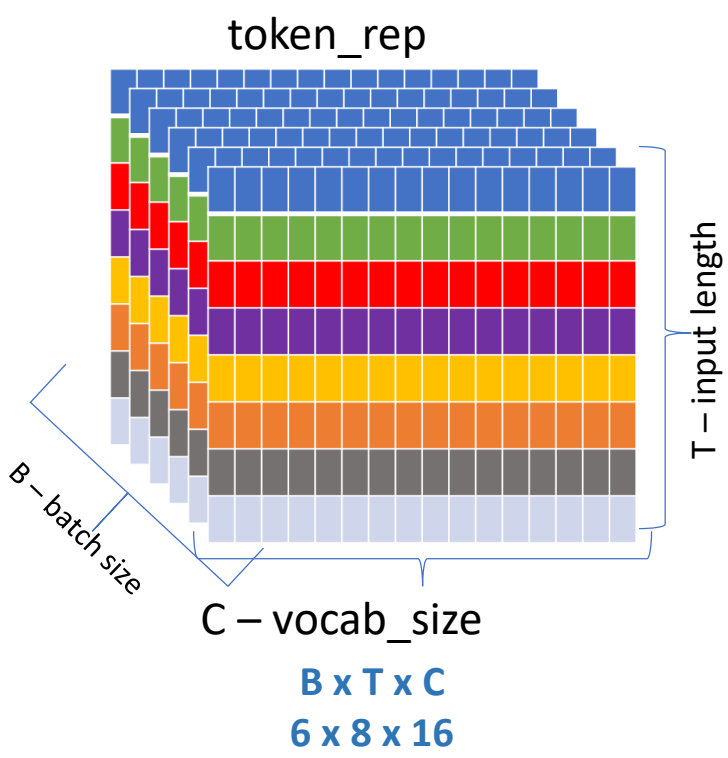


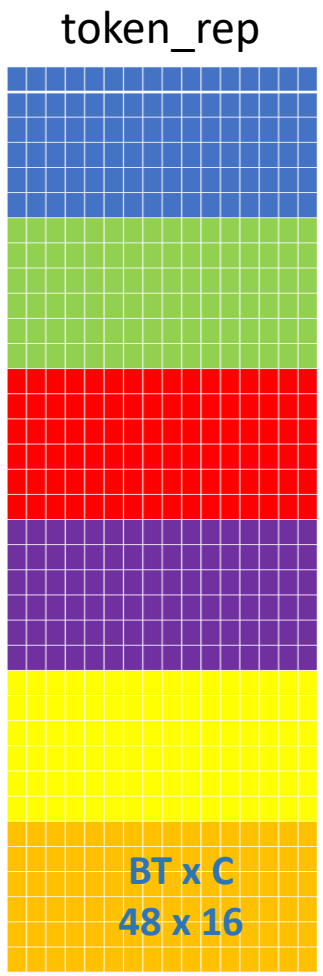
Forward pass

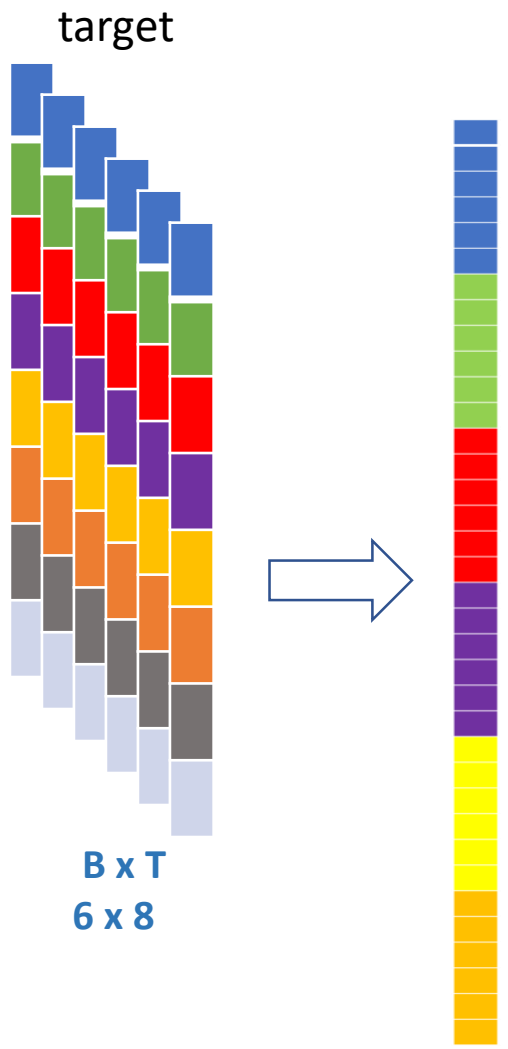
```
def forward(self, token, target=None):
    token_rep = self.token_embeddings_table(token)
    if target is None:
        ce_loss = None
    else:
        batch_size, input_length, vocab_size = token_rep.shape
        token_rep = token_rep.view(batch_size * input_length, vocab_size)
        targets = target.view(batch_size * input_length)
        ce_loss = F.cross_entropy(token_rep, targets)
    return token_rep, ce_loss
```

What comes next ?

Torch.cross_entropy(token_rep, target)







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

targets = target.view(batch_size * input_length)

Fit the model

```python
def fit(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=.01)
    for iteration in range(self.train_iters):
        if iteration % (self.train_iters//20) == 0:
            avg_loss = self.eval_loss()
            print(f"iter {iter} 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
```

Evaluate 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 infe
    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 t
            _, ce_loss = self(tokens, targets)  # forward pass
            losses[k] = ce_loss.item()  # the value of loss tensor as a standard Python
        perf[split] = losses.mean()
    self.train()  # turn-on training mode-
    return perf
```

Generate

```python
def generate(self, context_tokens, max_new_tokens):
    for _ in range(max_new_tokens):
        token_rep, _ = self(context_tokens)
        last_token_rep = token_rep[:, -1, :]
        probs = F.softmax(last_token_rep, dim=1)
        next_token = torch.multinomial(probs, num_samples=1)
        context_tokens = torch.cat((context_tokens, next_token), dim=1)
    output_text = self.decoder(context_tokens[0].tolist())
    return output_text
```

```
''' Look at the very last character to generate next
    @Author: Uzair Ahmad
    2022
'''
class BigramLanguageModel(nn.Module):
    def __init__(self, batch_size=4, input_length=8, train_iters=100, eval_iters=100):...
    def forward(self, token, target=None):...
    def fit(self):...
    def generate(self, context_tokens, max_new_tokens):...
    @torch.no_grad()  # tell torch not to prepare for back-propagation
    def eval_loss(self):...
    def prep(self, text):...
    def get_batch(self, split='train'):...
```