bert

March 14, 2022

[1]:
```python
import sys
import argparse
import numpy as np
from collections import defaultdict
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils import data
import torch.optim as optim
import einops
from tqdm import tqdm
import time
```

[2]:
```python
class Sequences(data.Dataset):
    '''This class reads in the sequences, extract the alphabet'''

    def __init__(self, filename, mask_prob):
        super(Sequences, self).__init__()
        '''read protein sequences from file'''
        self.mask_prob = mask_prob  # masking probability
        self.sequences = []  # set of sequences
        self.alphabet = {}  # set of characters/symbols
        with open(filename, "r") as f:
            for line in f.readlines():
                # skip lines beginning with # or "sequence"
                a = line.strip()
                if a[0] == "#":
                    continue
                elif a == "sequence":
                    continue
                else:
                    self.sequences.append(a)
                    for c in a:
                        self.alphabet[c] = True

        # find the max seq len and set the block size for transformer
        self.block_size = max([len(s) for s in self.sequences])
        self.block_size += 1  # first token is always CLS
```

```python
        print("BLOCK SIZE", self.block_size)

        # distinct chars/AA
        self.alphabet = sorted(self.alphabet.keys())
        self.alphabet_idx = {aa: i for i, aa in enumerate(self.alphabet)}

        # add special non-alphabet tokens
        self.a_sz = len(self.alphabet) # orig alphabet size, without extra tokens
        self.alphabet_idx['MASK'] = self.a_sz
        self.alphabet_idx['CLS'] = self.a_sz + 1
        self.alphabet_idx['PAD'] = self.a_sz + 2

    def __len__(self):
        '''return number of sequences'''
        return len(self.sequences)

    def __getitem__(self, idx):
        '''tokenize the sequence at idx -- one token per AA
        make sure all sequences are padded to block_size
        add CLS to front and PAD tokens if sequence is short
        replace masked positions with MASK
        return tokenized sequence, mask array, and true labels/tokens'''
        S = self.sequences[idx]
        # add CLS to front
        tokenized_seq = [self.alphabet_idx['CLS']]
        # actual AA sequence
        tokenized_seq.extend([self.alphabet_idx[S[i]] for i in range(len(S))])
        # PAD as remaining elements
        pad_len = self.block_size - len(tokenized_seq)
        tokenized_seq.extend([self.alphabet_idx['PAD'] for i in range(pad_len)])

        tokenized_seq = np.array(tokenized_seq)
        labels = tokenized_seq.copy()  # labels are same as original tokens

        # MASK out random positions given by mask_prob
        # notice that PAD positions are never masked
        num_masked = int(len(S) * self.mask_prob)
        mask_idxs = np.random.choice(len(S), num_masked, replace=False)
        mask_idxs += 1  # offset by 1 since CLS is 1st token

        # create mask array for masked positions
        mask_ary = np.zeros(self.block_size, dtype=int)
        mask_ary[mask_idxs] = 1

        # 10% of the mask_idxs will be left as is, without replacing with MASK
        # or random token. So select the 90% of remaining idxs
```

25 26 27

0 . . . 24 ← actual AA

$l = 1000$

$O(l^2)$!

labels = [CLS | s[1] . . . s[L] | PAD | . . . | PAD]

[0, 5, 15, . .]

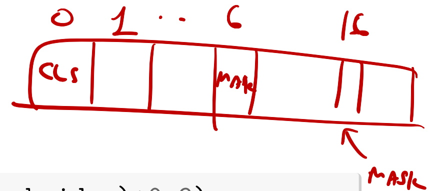
1 6 16

1000

[0 | 1 | . . . | 1 | . . . | 1 | . . . 0]

1 6 16

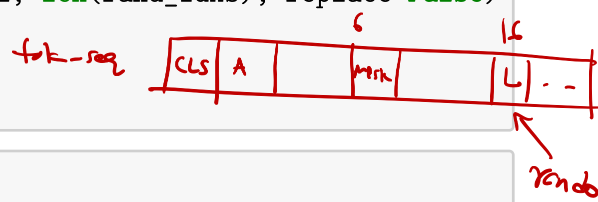
1. orig input 10%
2. MASK 80%
3. random symbol 10%

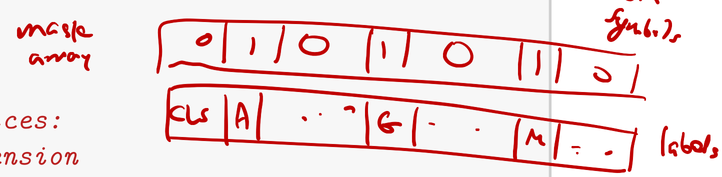



```
        mask_idxs = np.random.choice(mask_idxs, int(len(mask_idxs)*0.9), replace=False)
```

[6 16]

```
        # replace remaining mask_idxs with MASK token
        tokenized_seq[mask_idxs] = self.alphabet_idx['MASK']

        # select 10% of mask idxs to replace with random token
        rand_idxs = np.random.choice(mask_idxs, int(len(mask_idxs)*0.1), replace=False)
        # ideally replace proportional to token frequency, but we'll do it at random
        rand_tokens = np.random.choice(self.a_sz, len(rand_idxs), replace=False)
        tokenized_seq[rand_idxs] = rand_tokens

        return tokenized_seq, mask_ary, labels
```





[3]:
```
class SelfAttention(nn.Module):
    '''Self Attention'''

    def __init__(self, d, dk):
        '''define WQ, WK, WV projection matrices:
        d: d_model is the original model dimension
        dk: projection dimension for query, keys and values
        '''
        super(SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.WQ = nn.Linear(self.d, self.dk, bias=False)
        self.WK = nn.Linear(self.d, self.dk, bias=False)
        self.WV = nn.Linear(self.d, self.dk, bias=False)

    def forward(self, x):
        '''project the context onto key, query and value spaces and
        return the final value vectors
        '''
```

(b, ℓ, d)

```
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h
        Q = self.WQ(x)  # shape: b, l, dk
        K = self.WK(x)  # shape: b, l, dk
        V = self.WV(x)  # shape: b, l, dk

        K = torch.transpose(K, 1, 2)  # K.T transpose
        QKT = torch.bmm(Q, K)  # shape: b, l, l
```

$\begin{bmatrix} b \times \ell \times d_k \\ b \times d_k \times \ell \end{bmatrix} \begin{array}{l} Q \\ \leftarrow K.T \end{array}$

```
        # attention matrix
        # row specifies weights for the value vectors, row add up to one
        A = F.softmax(QKT / np.sqrt(self.dk), dim=2)  # shape: b, l, l
```

$V = b \times \underline{l} \times d_k$

$\frac{Q \cdot K^T}{\sqrt{d_k}} = b \times l \times l$

0 1 2

$l = 3$

0.1	0.5	0.4

```python
        V = torch.bmm(A, V)  # shape: b, l, dk
        return V


class SepHeads_SelfAttention(nn.Module):
    '''Separate Headed Self Attention: List of Attention Heads
    This is a straightforward implementation of the multiple heads.
    We have separate WQ, WK and WV matrices, one per head.'''

    def __init__(self, d, dk, num_heads):
        '''create separate heads:
        d: d_model dimension
        dk: projection dimension for query, keys and values
        num_heads: number of attention heads
        '''
        super(SepHeads_SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.num_heads = num_heads  # number of attention heads

        self.sa_layers = nn.ModuleList()
        for i in range(self.num_heads):
            self.sa_layers.append(SelfAttention(self.d, self.dk))

        self.WO = nn.Linear(self.dk * self.num_heads, self.d, bias=False)

    def forward(self, x):
        '''use separate attention heads, and concat values'''
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h
        V = []
        for i in range(self.num_heads):
            V.append(self.sa_layers[i](x))

        # concat all the value vectors from the heads
        V = torch.cat(V, dim=2)  # shape: b, l, h x dk
        # project back to d_model
        x = self.WO(V)  # shape: b, l, d
        return x


class MultiHead_SelfAttention(nn.Module):
    '''Multi Headed Self Attention:
    Instead of using a list of attention heads with separate WQ, WK, WV␣
↪matrices,
    we combine all heads into one, and use a single WQ, WK and WV matrix.
```

0 1 2

h0 : $b \times l \times d_k$

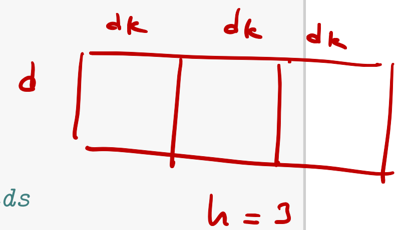
h1 :

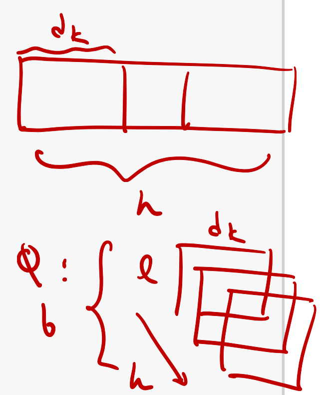
.
.

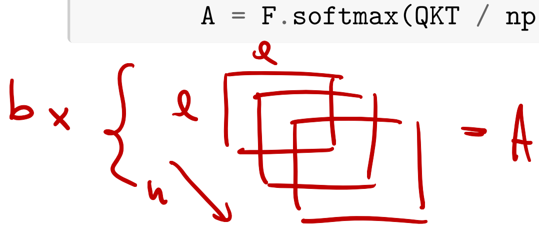
h8 :

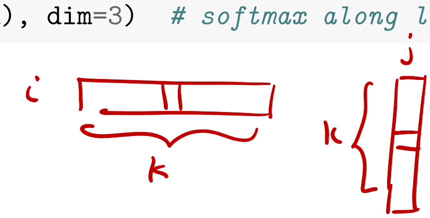
$b \times l \times (d_k \cdot h)$

```python
    Each matrix maps the d-dim input block into h*dk dim space, where h is num_heads.
    We have to carefully keep the heads separate for softmax to achieve the same
    effect at from the list of heads. We do that via einops and the very useful
    torch.einsum function.

    This function is much more efficient than using separate heads.
    '''

    def __init__(self, d, dk, num_heads):
        '''create multi-heads -- joint heads:
        d: d_model dimension
        dk: projection dimension for query, keys and values
        num_heads: number of attention heads
        '''
        super(MultiHead_SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.num_heads = num_heads  # number of attention heads

        self.WQ = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WK = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WV = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WO = nn.Linear(self.dk * self.num_heads, self.d, bias=False)

    def forward(self, x):
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h, d_model=d
        Q = self.WQ(x)  # size: (b, l, h*dk)
        K = self.WK(x)  # size: (b, l, h*dk)
        V = self.WV(x)  # size: (b, l, h*dk)

        # split Q, K, V into heads and dk, move heads up front; KT is transpose of K
        Q = einops.rearrange(
            Q, 'b l (h dk)-> b h l dk', h=self.num_heads
        )  # size: (b, h, l, dk)
        KT = einops.rearrange(
            K, 'b l (h dk)-> b h dk l', h=self.num_heads
        )  # size: (b, h, dk, l)
        V = einops.rearrange(
            V, 'b l (h dk)-> b h l dk', h=self.num_heads
        )  # size: (b, h, l, dk)

        # compute Q x K.T, output is (b, h, l, l)
        QKT = torch.einsum('bhik,bhkj->bhij', Q, KT)
        A = F.softmax(QKT / np.sqrt(self.dk), dim=3)  # softmax along last dim
```



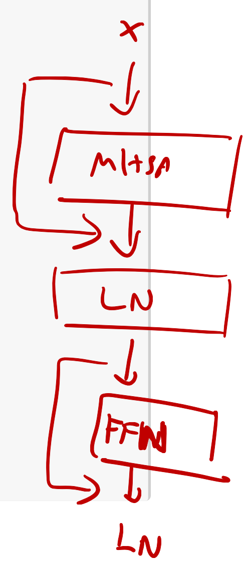




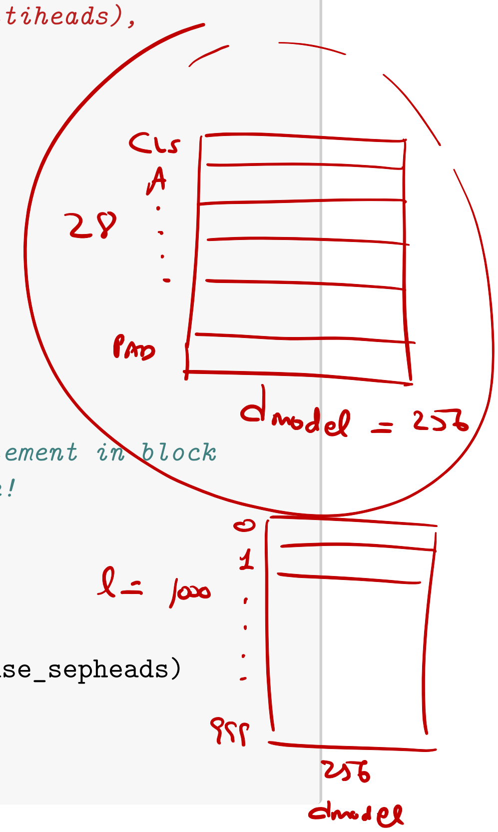


```
        # new value representation
        V = torch.einsum('bhik,bhkj->bhij', A, V)  # size: (b, h, l, dk)
        V = einops.rearrange(V, 'b h l dk -> b l (h dk)')  # size: (b, l, h*dk)

        # shape: b, l, h x dk
        x = self.WO(V)  # shape: b, l, d
        return x


class TransformerBlock(nn.Module):
    '''Transformer Block: multi-head or separate heads of attention,
    followed by layernorm, ffn, and another layernorm
    '''

    def __init__(self, d, dk, num_heads, block_size, use_sepheads):
        '''
        d: d_model dimension
        dk: projection dimension
        num_heads: number of attention heads
        use_sepheads: use separate heads or multiheads,
                      multiheads is much more efficient
        '''
        super(TransformerBlock, self).__init__()
        self.use_sepheads = use_sepheads
        self.drop_prob = 0.1

        if self.use_sepheads:
            # uses for loop for separate heads
            self.mhsa = SepHeads_SelfAttention(d, dk, num_heads)
        else:
            # this is more efficient
            self.mhsa = MultiHead_SelfAttention(d, dk, num_heads)

        self.ln1 = nn.LayerNorm(d)  # layer norm
        self.ffn = nn.Sequential( #FFN module
            nn.Linear(d, d),  # linear layer
            nn.ReLU(),  # relu
            nn.Linear(d, d)  # linear layer
        )
        self.ln2 = nn.LayerNorm(d)  # layer norm

    def forward(self, x):
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h, d_model=d
        x_sa = self.mhsa(x)  # multiple attention heads
        x_sa = F.dropout(x_sa, p=self.drop_prob)
```

bxl x(d)



```python
        x_ln1 = self.ln1(x + x_sa)  # residual layer + layer norm
        # two linear layers with relu in between
        x_ffn = self.ffn(x_ln1)
        x_ffn = F.dropout(x_ffn, p=self.drop_prob)
        x_ln2 = self.ln2(x_ln1 + x_ffn)  # residual layer + layer norm
        return x_ln2


class Transformer(nn.Module):
    '''Transformer model:
    input is a block of tokens: first token is always CLS
    MASK token for positions for training the masked language model
    PAD tokens at end for sequences shorter than block size'''

    def __init__(
        self, d, dk, block_size, num_layers, num_heads, alphabet_idx, use_sepheads
    ):
        '''
        d: d_model dimension
        dk: projection dimension
        block_size: the max sequence length
        num_layers: how many transformer blocks/layers?
        num_heads: number of attention heads
        alphabet_idx: dict of tokens to token ids (ints)
        use_sepheads: use separate heads or joint heads (multiheads),
                      multiheads is much more efficient

        '''
        super(Transformer, self).__init__()
        self.num_layers = num_layers
        self.drop_prob = 0.1  # for dropout layer

        # embedding layer to map tokens to d dim vectors
        self.embed = nn.Embedding(len(alphabet_idx), d, padding_idx=alphabet_idx['PAD'])

        # learnable position embeddings, one per sequence element in block
        # can also use sine/cosine embeddings: not done here!
        self.pos_embed = nn.Embedding(block_size, d)

        # list of transformer blocks/layers
        tb_list = [
            TransformerBlock(d, dk, num_heads, block_size, use_sepheads)
            for i in range(self.num_layers)
        ]
        # combine all layers into one "sequential" layer
```



y = [0 1 2 . . . - - - 599]

p = pos-embed(y)

1 c

[CLS | A | . . . | MASK | - - -]

⇓ embed

```python
        self.layers = nn.Sequential(*tb_list)

    def forward(self, x):
        # input shape: batch_size (b), block_size (l)
        # d is d_model
        p = self.pos_embed.weight  # shape: l, d
        x = self.embed(x) + p  # add pos embeddings, shape: b, l, d
        x = F.dropout(x, p=self.drop_prob)  # dropout
        x = self.layers(x)  # shape: (b, l, d)
        return x
```

b x l x d output

pos : l x d

```python
class BERT(nn.Module):
    '''BERT model'''

    def __init__(
        self,
        d,  # d_model
        dk,  # projection dimension for queries, keys and values
        block_size,  # max sequence length
        num_layers,  # number of transformer layers
        num_heads,  # number of attention heads
        alphabet_idx,  # mapping from alphabet/token to id
        alphabet,  # alphabet/tokens (all AA + PAD + MASK + CLS)
        use_sepheads,  # use separate or joint attention heads
    ):
        super(BERT, self).__init__()
        self.d = d
        self.dk = dk
        self.block_size = block_size
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.use_sepheads = use_sepheads
        self.alphabet_idx = alphabet_idx

        # transformer model
        self.transformer = Transformer(
            d, dk, block_size, num_layers, num_heads, alphabet_idx, use_sepheads
        )

        # map transformer model to one of the tokens (for classification)
        self.linear = nn.Linear(d, len(alphabet))

    def forward(self, x):
        # input shape: batch_size (b), block_size (l)
        x = self.transformer(x)  # shape: b, l, d
        x = self.linear(x)  # shape: b, l, #tokens
```

b x l x 28

output of BERT

```python
        return x

    def save_transformer(self, args):
        '''save the transformer portion'''
        fname = f'transformer_d{self.d}_dk{self.dk}_l{self.num_layers}'
        fname += f'_h{self.num_heads}_lr{args.learning_rate}'
        fname += f'_e{args.epochs}_j{args.jobid}.pth'
        saveinfo = {
            'd': self.d,
            'dk': self.dk,
            'l': self.num_layers,
            'h': self.num_heads,
            'sh': self.use_sepheads,
            'block_size': self.block_size,
            'alphabet_idx': self.alphabet_idx,
            'model': self.transformer.state_dict(),
        }
        torch.save(saveinfo, fname)

    def checkpoint(self, args, bidx, e, running_loss, optimizer):
        '''check point the model and optimizer states
        useful to resume training later'''
        ckpt_fname = f'ckpt_J{args.jobid}.pth'
        checkpoint = {
            'd': self.d,
            'dk': self.dk,
            'l': self.num_layers,
            'h': self.num_heads,
            'alphabex_idx': self.alphabet_idx,
            'batch': bidx,
            'epoch': e,
            'loss': running_loss,
            'state_dict': self.state_dict(),
            'optimizer': optimizer.state_dict(),
        }
        torch.save(checkpoint, ckpt_fname)
```

[6]:
```python
def parse_args():
    parser = argparse.ArgumentParser(description='bert.py')
    parser.add_argument('-f', dest='fname')
    parser.add_argument('-d', default=256, type=int) #d_model
    parser.add_argument('-dk', default=32, type=int) #d_k
    parser.add_argument('-l', dest = 'num_layers', default=1, type=int)
    parser.add_argument('-H', dest = 'num_heads', default=8, type=int)
    parser.add_argument('-sH', dest = 'use_sepheads', default = False,
                            action='store_true') # use separate heads?
    parser.add_argument('-m', dest='mask_prob', default=0.15, type=float)
```

```
    parser.add_argument('-e', dest='epochs', default=10, type=int)
    parser.add_argument('-nw', dest='num_workers', default=0, type=int)
    parser.add_argument('-b', dest='batch_size', default=4, type=int)
    parser.add_argument('-lr', dest='learning_rate', default=0.01, type=float)
    parser.add_argument('-wd', dest='weight_decay', default=0, type=float)
    parser.add_argument('-j', dest='jobid', default=1, type=int)
    parser.add_argument('-D', dest='device', default=0, type=int)
    parser.add_argument('-c', dest='chkpt_fname', default=None) # if given,␣
 ↪resume from checkpoint

    # set the input args for running the code
    cmd = "-f small_uniprot.txt "
    cmd += "-lr 1e-4 -wd 1e-7 -e 10"
    #cmd += "-c ckpt_J1.pth"
    args = parser.parse_args(cmd.split())

    return args
```

[7]:

```
# Main training wrapper code
args = parse_args()
print(args)

if torch.cuda.is_available():
    device = f"cuda:{args.device}"
    print("using device", torch.cuda.get_device_name(device))
else:
    device = "cpu"

# read sequences, create dataset
S = Sequences(args.fname, args.mask_prob)
data_gen = data.DataLoader(S,
                           batch_size=args.batch_size,
                           num_workers=args.num_workers, shuffle=True)

# create the BERT model
model = BERT(args.d, args.dk, S.block_size, args.num_layers,
                    args.num_heads, S.alphabet_idx, S.alphabet, args.
 ↪use_sepheads)

# use suggested transformer betas
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate,
                       betas = (0.9, 0.98),
                       weight_decay=args.weight_decay)

prev_e = 0 # epoch number
if args.chkpt_fname is not None:
    saveinfo = torch.load(args.chkpt_fname)
```

```python
    prev_e = saveinfo['epoch']+1
    model.load_state_dict(saveinfo['state_dict'])
    optimizer.load_state_dict(saveinfo['optimizer'])
    print("resume from epoch:", prev_e) # resume from prev_e

model = model.to(device)
model.train()
```

```
Namespace(fname='small_uniprot.txt', d=256, dk=32, num_layers=1, num_heads=8,
use_sepheads=False, mask_prob=0.15, epochs=10, num_workers=0, batch_size=4,
learning_rate=0.0001, weight_decay=1e-07, jobid=1, device=0, chkpt_fname=None)
using device NVIDIA GeForce RTX 3090
BLOCK SIZE 990
```

[7]:
```
BERT(
  (transformer): Transformer(
    (embed): Embedding(27, 256, padding_idx=26)
    (pos_embed): Embedding(990, 256)
    (layers): Sequential(
      (0): TransformerBlock(
        (mhsa): MultiHead_SelfAttention(
          (WQ): Linear(in_features=256, out_features=256, bias=False)
          (WK): Linear(in_features=256, out_features=256, bias=False)
          (WV): Linear(in_features=256, out_features=256, bias=False)
          (WO): Linear(in_features=256, out_features=256, bias=False)
        )
        (ln1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
        (ffn): Sequential(
          (0): Linear(in_features=256, out_features=256, bias=True)
          (1): ReLU()
          (2): Linear(in_features=256, out_features=256, bias=True)
        )
        (ln2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      )
    )
  )
  (linear): Linear(in_features=256, out_features=24, bias=True)
)
```

[8]:
```python
# usual boilerplate training loop over epochs
start_t = time.time()
for e in range(prev_e, prev_e + args.epochs):
    running_loss = 0.
    correct = 0.
    num_masked = 0.
    for bidx, (block, mask, labels) in enumerate(tqdm(data_gen)):
        block = block.to(device)
        mask = mask.to(device)
```

```
        labels = labels.to(device)

        model.zero_grad()

        # shape: b, l, C
        # where b is batch_size, l is block_size, C is number of classes
        preds = model(block)

        #cross_entropy expects b, C, l
        preds = preds.swapaxes(1,2) # shape: b, C, l

        # retain loss per position, since we will zero out non-mask positions
        loss = F.cross_entropy(preds, labels, reduction='none')
        loss = torch.sum(loss*mask)

        loss.backward()
        optimizer.step()
        running_loss += loss.item()

        # compute number of correct predictions, keep track of num_masked for
 ↪acc
        pred_labels = torch.argmax(preds, dim=1)
        correct += torch.sum((pred_labels == labels)*mask).item()
        num_masked += torch.sum(torch.where(mask == 1)[0]).item()

        # checkpoint every 100 batches
        if bidx % 100 == 0:
            model.checkpoint(args, bidx, e, running_loss, optimizer)

    # print loss at end of each epoch
    acc = correct/num_masked
    print("epoch", e, running_loss, bidx, running_loss / (bidx + 1), acc,
 ↪correct, num_masked)
    model.checkpoint(args, bidx, e, running_loss, optimizer)

# save the transformer model for downstream classification
model.save_transformer(args)

end_t = time.time()
print("finished in time", end_t - start_t, args.num_workers)
```

```
100%|                                      |
250/250 [00:02<00:00, 98.34it/s]

epoch 0 143421.14875793457 249 573.6845950317382 0.0893734847180085 6746.0
75481.0

100%|                                      |
```

```
250/250 [00:02<00:00, 107.09it/s]

epoch 1 134677.72203063965 249 538.7108881225586 0.12095391211146839 9028.0
74640.0

100%|                                     |
250/250 [00:02<00:00, 107.68it/s]

epoch 2 133828.6978149414 249 535.3147912597656 0.12414963713395971 9289.0
74821.0

100%|                                     |
250/250 [00:02<00:00, 106.82it/s]

epoch 3 133506.07514953613 249 534.0243005981446 0.12491376107838455 9415.0
75372.0

100%|                                     |
250/250 [00:02<00:00, 106.59it/s]

epoch 4 133231.82731628418 249 532.9273092651367 0.12503824307966532 9400.0
75177.0

100%|                                     |
250/250 [00:02<00:00, 108.05it/s]

epoch 5 133032.2057647705 249 532.128823059082 0.12536586011892936 9466.0
75507.0

100%|                                     |
250/250 [00:02<00:00, 107.59it/s]

epoch 6 132769.43391418457 249 531.0777356567382 0.12696506258841114 9514.0
74934.0

100%|                                     |
250/250 [00:02<00:00, 107.64it/s]

epoch 7 132619.56163024902 249 530.4782465209961 0.12879863110002784 9710.0
75389.0

100%|                                     |
250/250 [00:02<00:00, 105.48it/s]

epoch 8 132590.02340698242 249 530.3600936279297 0.12933135530359224 9764.0
75496.0

100%|                                     |
250/250 [00:02<00:00, 107.22it/s]

epoch 9 132367.76065063477 249 529.4710426025391 0.13242138875358553 9787.0
73908.0
finished in time 23.862635374069214 0
```

```python
#!/usr/bin/env python
# coding: utf-8

import os

# import hostlist
import argparse
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils import data
import torch.optim as optim
import torch.distributed as dist
from torch.utils.data.distributed import DistributedSampler
import einops
from tqdm import tqdm
import time


class Sequences(data.Dataset):
    '''This class reads in the sequences, extract the alphabet'''

    def __init__(self, filename, mask_prob):
        super(Sequences, self).__init__()
        '''read protein sequences from file'''
        self.mask_prob = mask_prob  # masking probability
        self.sequences = []  # set of sequences
        self.alphabet = {}  # set of characters/symbols
        with open(filename, "r") as f:
            for line in f.readlines():
                # skip lines beginning with # or "sequence"
                a = line.strip()
                if a[0] == "#":
                    continue
                elif a == "sequence":
                    continue
                else:
                    self.sequences.append(a)
                    for c in a:
                        self.alphabet[c] = True

        # find the max seq len and set the block size for transformer
        self.block_size = max([len(s) for s in self.sequences])
        self.block_size += 1  # first token is always CLS
        print("BLOCK SIZE", self.block_size)

        # distinct chars/AA
        self.alphabet = sorted(self.alphabet.keys())
        self.alphabet_idx = {aa: i for i, aa in enumerate(self.alphabet)}

        # add special non-alphabet tokens
        self.a_sz = len(self.alphabet)  # orig alphabet size, without extra tokens
        self.alphabet_idx['MASK'] = self.a_sz
        self.alphabet_idx['CLS'] = self.a_sz + 1
        self.alphabet_idx['PAD'] = self.a_sz + 2

    def __len__(self):
        '''return number of sequences'''
        return len(self.sequences)

    def __getitem__(self, idx):
        '''tokenize the sequence at idx -- one token per AA
        make sure all sequences are padded to block_size
        add CLS to front and PAD tokens if sequence is short
        replace masked positions with MASK
        return tokenized sequence, mask array, and true labels/tokens'''
        S = self.sequences[idx]
        # add CLS to front
        tokenized_seq = [self.alphabet_idx['CLS']]
        # actual AA sequence
        tokenized_seq.extend([self.alphabet_idx[S[i]] for i in range(len(S))])
        # PAD as remaining elements
```

```
        pad_len = self.block_size - len(tokenized_seq)
        tokenized_seq.extend([self.alphabet_idx['PAD'] for i in range(pad_len)])

        tokenized_seq = np.array(tokenized_seq)
        labels = tokenized_seq.copy()  # labels are same as original tokens

        # MASK out random positions given by mask_prob
        # notice that PAD positions are never masked
        num_masked = int(len(S) * self.mask_prob)
        mask_idxs = np.random.choice(len(S), num_masked, replace=False)
        mask_idxs += 1  # offset by 1 since CLS is 1st token

        # create mask array for masked positions
        mask_ary = np.zeros(self.block_size, dtype=int)
        mask_ary[mask_idxs] = 1

        # 10% of the mask_idxs will be left as is, without replacing with MASK
        # or random token. So select the 90% of remaining idxs
        mask_idxs = np.random.choice(
            mask_idxs, int(len(mask_idxs) * 0.9), replace=False
        )

        # replace remaining mask_idxs with MASK token
        tokenized_seq[mask_idxs] = self.alphabet_idx['MASK']

        # select 10% of mask idxs to replace with random token
        rand_idxs = np.random.choice(
            mask_idxs, int(len(mask_idxs) * 0.1), replace=False
        )
        # ideally replace proportional to token frequency, but we'll do it at random
        rand_tokens = np.random.choice(self.a_sz, len(rand_idxs), replace=False)
        tokenized_seq[rand_idxs] = rand_tokens

        return tokenized_seq, mask_ary, labels


class SelfAttention(nn.Module):
    '''Self Attention'''

    def __init__(self, d, dk):
        '''define WQ, WK, WV projection matrices:
        d: d_model is the original model dimension
        dk: projection dimension for query, keys and values
        '''
        super(SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.WQ = nn.Linear(self.d, self.dk, bias=False)
        self.WK = nn.Linear(self.d, self.dk, bias=False)
        self.WV = nn.Linear(self.d, self.dk, bias=False)

    def forward(self, x):
        '''project the context onto key, query and value spaces and
        return the final value vectors
        '''
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h
        Q = self.WQ(x)  # shape: b, l, dk
        K = self.WK(x)  # shape: b, l, dk
        V = self.WV(x)  # shape: b, l, dk

        K = torch.transpose(K, 1, 2)  # K.T transpose
        QKT = torch.bmm(Q, K)  # shape: b, l, l

        # attention matrix
        # row specifies weights for the value vectors, row add up to one
        A = F.softmax(QKT / np.sqrt(self.dk), dim=2)  # shape: b, l, l

        V = torch.bmm(A, V)  # shape: b, l, dk
        return V


class SepHeads_SelfAttention(nn.Module):
```

```
    '''Separate Headed Self Attention: List of Attention Heads
    This is a straightforward implementation of the multiple heads.
    We have separate WQ, WK and WV matrices, one per head.'''

    def __init__(self, d, dk, num_heads):
        '''create separate heads:
        d: d_model dimension
        dk: projection dimension for query, keys and values
        num_heads: number of attention heads
        '''
        super(SepHeads_SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.num_heads = num_heads  # number of attention heads

        self.sa_layers = nn.ModuleList()
        for i in range(self.num_heads):
            self.sa_layers.append(SelfAttention(self.d, self.dk))

        self.WO = nn.Linear(self.dk * self.num_heads, self.d, bias=False)

    def forward(self, x):
        '''use separate attention heads, and concat values'''
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h
        V = []
        for i in range(self.num_heads):
            V.append(self.sa_layers[i](x))

        # concat all the value vectors from the heads
        V = torch.cat(V, dim=2)  # shape: b, l, h x dk
        # project back to d_model
        x = self.WO(V)  # shape: b, l, d
        return x


class MultiHead_SelfAttention(nn.Module):
    '''Multi Headed Self Attention:
    Instead of using a list of attention heads with separate WQ, WK, WV matrices,
    we combine all heads into one, and use a single WQ, WK and WV matrix.
    Each matrix maps the d-dim input block into h*dk dim space, where h is num_heads
.
    We have to carefully keep the heads separate for softmax to achieve the same
    effect at from the list of heads. We do that via einops and the very useful
    torch.einsum function.

    This function is much more efficient than using separate heads.
    '''

    def __init__(self, d, dk, num_heads):
        '''create multi-heads -- joint heads:
        d: d_model dimension
        dk: projection dimension for query, keys and values
        num_heads: number of attention heads
        '''
        super(MultiHead_SelfAttention, self).__init__()
        self.d = d  # d_model
        self.dk = dk  # d_k: projection dimension
        self.num_heads = num_heads  # number of attention heads

        self.WQ = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WK = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WV = nn.Linear(self.d, self.dk * self.num_heads, bias=False)
        self.WO = nn.Linear(self.dk * self.num_heads, self.d, bias=False)

    def forward(self, x):
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h, d_model=d
        Q = self.WQ(x)  # size: (b, l, h*dk)
        K = self.WK(x)  # size: (b, l, h*dk)
        V = self.WV(x)  # size: (b, l, h*dk)

        # split Q, K, V into heads and dk, move heads up front; KT is transpose of K
```

```
        Q = einops.rearrange(
            Q, 'b l (h dk)-> b h l dk', h=self.num_heads
        )  # size: (b, h, l, dk)
        KT = einops.rearrange(
            K, 'b l (h dk)-> b h dk l', h=self.num_heads
        )  # size: (b, h, dk, l)
        V = einops.rearrange(
            V, 'b l (h dk)-> b h l dk', h=self.num_heads
        )  # size: (b, h, l, dk)

        # compute Q x K.T, output is (b, h, l, l)
        QKT = torch.einsum('bhik,bhkj->bhij', Q, KT)
        A = F.softmax(QKT / np.sqrt(self.dk), dim=3)  # softmax along last dim

        # new value representation
        V = torch.einsum('bhik,bhkj->bhij', A, V)  # size: (b, h, l, dk)
        V = einops.rearrange(V, 'b h l dk -> b l (h dk)')  # size: (b, l, h*dk)

        # shape: b, l, h x dk
        x = self.WO(V)  # shape: b, l, d
        return x


class TransformerBlock(nn.Module):
    '''Transformer Block: multi-head or separate heads of attention,
    followed by layernorm, ffn, and another layernorm
    '''

    def __init__(self, d, dk, num_heads, block_size, use_sepheads):
        '''
        d: d_model dimension
        dk: projection dimension
        num_heads: number of attention heads
        use_sepheads: use separate heads or multiheads,
                      multiheads is much more efficient
        '''
        super(TransformerBlock, self).__init__()
        self.use_sepheads = use_sepheads
        self.drop_prob = 0.1

        if self.use_sepheads:
            # uses for loop for separate heads
            self.mhsa = SepHeads_SelfAttention(d, dk, num_heads)
        else:
            # this is more efficient
            self.mhsa = MultiHead_SelfAttention(d, dk, num_heads)

        self.ln1 = nn.LayerNorm(d)  # layer norm
        self.ffn = nn.Sequential(  # FFN module
            nn.Linear(d, d),  # linear layer
            nn.ReLU(),  # relu
            nn.Linear(d, d),  # linear layer
        )
        self.ln2 = nn.LayerNorm(d)  # layer norm

    def forward(self, x):
        # input shape: (batch_size, block_size, d)
        # let batch_size=b, block_size=l, num_heads=h, d_model=d
        x_sa = self.mhsa(x)  # multiple attention heads
        x_sa = F.dropout(x_sa, p=self.drop_prob)
        x_ln1 = self.ln1(x + x_sa)  # residual layer + layer norm
        # two linear layers with relu in between
        x_ffn = self.ffn(x_ln1)
        x_ffn = F.dropout(x_ffn, p=self.drop_prob)
        x_ln2 = self.ln2(x_ln1 + x_ffn)  # residual layer + layer norm
        return x_ln2


class Transformer(nn.Module):
    '''Transformer model:
    input is a block of tokens: first token is always CLS
    MASK token for positions for training the masked language model
    PAD tokens at end for sequences shorter than block size'''
```

```
    def __init__(
        self, d, dk, block_size, num_layers, num_heads, alphabet_idx, use_sepheads
    ):
        '''
        d: d_model dimension
        dk: projection dimension
        block_size: the max sequence length
        num_layers: how many transformer blocks/layers?
        num_heads: number of attention heads
        alphabet_idx: dict of tokens to token ids (ints)
        use_sepheads: use separate heads or joint heads (multiheads),
                      multiheads is much more efficient

        '''
        super(Transformer, self).__init__()
        self.num_layers = num_layers
        self.drop_prob = 0.1  # for dropout layer

        # embedding layer to map tokens to d dim vectors
        self.embed = nn.Embedding(len(alphabet_idx), d, padding_idx=alphabet_idx['PA
D'])

        # learnable position embeddings, one per sequence element in block
        # can also use sine/cosine embeddings: not done here!
        self.pos_embed = nn.Embedding(block_size, d)

        # list of transformer blocks/layers
        tb_list = [
            TransformerBlock(d, dk, num_heads, block_size, use_sepheads)
            for i in range(self.num_layers)
        ]
        # combine all layers into one "sequential" layer
        self.layers = nn.Sequential(*tb_list)

    def forward(self, x):
        # input shape: batch_size (b), block_size (l)
        # d is d_model
        p = self.pos_embed.weight  # shape: l, d
        x = self.embed(x) + p  # add pos embeddings, shape: b, l, d
        x = F.dropout(x, p=self.drop_prob)  # dropout
        x = self.layers(x)  # shape: (b, l, d)
        return x


class BERT(nn.Module):
    '''BERT model'''

    def __init__(
        self,
        d,  # d_model
        dk,  # projection dimension for queries, keys and values
        block_size,  # max sequence length
        num_layers,  # number of transformer layers
        num_heads,  # number of attention heads
        alphabet_idx,  # mapping from alphabet/token to id
        alphabet,  # alphabet/tokens (all AA + PAD + MASK + CLS)
        use_sepheads,  # use separate or joint attention heads
    ):
        super(BERT, self).__init__()
        self.d = d
        self.dk = dk
        self.block_size = block_size
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.use_sepheads = use_sepheads
        self.alphabet_idx = alphabet_idx

        # transformer model
        self.transformer = Transformer(
            d, dk, block_size, num_layers, num_heads, alphabet_idx, use_sepheads
        )
```

```
        # map transformer model to one of the tokens (for classification)
        self.linear = nn.Linear(d, len(alphabet))

    def forward(self, x):
        # input shape: batch_size (b), block_size (l)
        x = self.transformer(x)  # shape: b, l, d
        x = self.linear(x)  # shape: b, l, #tokens
        return x

    def save_transformer(self, args):
        '''save the transformer portion'''
        fname = f'transformer_d{self.d}_dk{self.dk}_l{self.num_layers}'
        fname += f'_h{self.num_heads}_lr{args.learning_rate}'
        fname += f'_e{args.epochs}_j{args.jobid}.pth'
        saveinfo = {
            'd': self.d,
            'dk': self.dk,
            'l': self.num_layers,
            'h': self.num_heads,
            'sh': self.use_sepheads,
            'block_size': self.block_size,
            'alphabet_idx': self.alphabet_idx,
            'model': self.transformer.state_dict(),
        }
        torch.save(saveinfo, fname)

    def checkpoint(self, args, bidx, e, running_loss, optimizer):
        '''check point the model and optimizer states
        useful to resume training later'''
        ckpt_fname = f'ckpt_J{args.jobid}.pth'
        checkpoint = {
            'd': self.d,
            'dk': self.dk,
            'l': self.num_layers,
            'h': self.num_heads,
            'alphabex_idx': self.alphabet_idx,
            'batch': bidx,
            'epoch': e,
            'loss': running_loss,
            'state_dict': self.state_dict(),
            'optimizer': optimizer.state_dict(),
        }
        torch.save(checkpoint, ckpt_fname)


def parse_args():
    parser = argparse.ArgumentParser(description='bert.py')
    parser.add_argument('-f', dest='fname')
    parser.add_argument('-d', default=256, type=int)  # d_model
    parser.add_argument('-dk', default=32, type=int)  # d_k
    parser.add_argument('-l', dest='num_layers', default=1, type=int)
    parser.add_argument('-H', dest='num_heads', default=8, type=int)
    parser.add_argument(
        '-sH', dest='use_sepheads', default=False, action='store_true'
    )  # use separate heads?
    parser.add_argument('-m', dest='mask_prob', default=0.15, type=float)
    parser.add_argument('-e', dest='epochs', default=10, type=int)
    parser.add_argument('-nw', dest='num_workers', default=0, type=int)
    parser.add_argument('-b', dest='batch_size', default=4, type=int)
    parser.add_argument('-lr', dest='learning_rate', default=0.01, type=float)
    parser.add_argument('-wd', dest='weight_decay', default=0, type=float)
    parser.add_argument('-j', dest='jobid', default=1, type=int)
    parser.add_argument('-D', dest='device', default=0, type=int)
    parser.add_argument('-c', dest='chkpt_fname', default=None)
    parser.add_argument('--local_rank', dest='localrank', default=-1, type=int)
    parser.add_argument(
        '--dist', dest='distrun', choices=['slurm', 'torchrun'], default=None
    )

    args = parser.parse_args()

    return args
```

Sbatch run.sh

```
def set_SLURM_vars(args):
    '''get SLURM variables

    # run as:
    #!/bin/bash
    # SBATCH --time=70
    # SBATCH --nodes=3
    # SBATCH --ntasks-per-node=6
    # SBATCH --gres=gpu:6
    # SBATCH --workdir=/gpfs/u/home/MLI2/MLI2zaki/scratch/Assign4
    # SBATCH --output=run_%j.out #append job id
    # SBATCH --mail-user=zaki@cs.rpi.edu
    # SBATCH --mail-type=BEGIN,END,FAIL
    ### init virtual environment if needed
    # conda activate base
    ### the command to run
    # srun bert-ddp-ccni.py -f uniprot-reviewed-lim_sequences.txt -l 2 -lr 1e-5 -wd
1e-7 -e 10
    #                                        --dist slurm -j "$SLURM_JOB_ID"
    '''
    args.rank = int(os.environ['SLURM_PROCID'])
    args.local_rank = int(os.environ['SLURM_LOCALID'])
    args.world_size = int(os.environ['SLURM_NTASKS'])

    # get node list from slurm
    # hostnames = hostlist.expand_hostlist(os.environ['SLURM_JOB_NODELIST'])

    # define MASTER_ADD & MASTER_PORT
    # os.environ['MASTER_ADDR'] = hostnames[0]
    os.environ['MASTER_ADDR'] = os.environ['SLURM_SUBMIT_HOST']
    os.environ['MASTER_PORT'] = "29500"  # to avoid port conflict on the same node
    args.master_addr = os.environ['MASTER_ADDR']


def set_TORCHRUN_vars(args):
    '''These are set by torchrun / torch.distributed.launch
    on DCS cluster use torch.distributed.launch

    # on DCS interactive, run as:
    python -m torch.distributed.launch --nproc_per_node=2 bert-ddp-ccni.py
              -f uniprot-reviewed-lim_sequences.txt -l 1 -lr 1e-4 -wd 1e-7 -e 1 --dist
torchrun
    # if torchrun available run as:
    torchrun --nproc_per_node=2 bert-ddp-ccni.py
                -f small_uniprot.txt -lr 1e-4 -wd 1e-7 -e 1 --dist torchrun
    '''
    args.rank = int(os.environ['RANK'])
    if args.localrank >= 0:
        args.local_rank = args.localrank
    else:
        args.local_rank = int(os.environ['LOCAL_RANK'])
    args.world_size = int(os.environ['WORLD_SIZE'])
    args.master_addr = os.environ['MASTER_ADDR']


def set_vars(args):
    '''sequential run

    # run as bert-ddp-ccni.py -f small_uniprot.txt -lr 1e-4 -wd 1e-7 -e 1
    '''
    args.rank = 0
    args.local_rank = 0
    args.world_size = 1
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'  # to avoid port conflict on the same node
    args.master_addr = os.environ['MASTER_ADDR']


def set_deterministic():
    torch.manual_seed(0)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

run.sh

3 machine ×6 18 GPUs

```
    np.random.seed(0)


# Main training wrapper code
if __name__ == "__main__":
    args = parse_args()

    if args.distrun == 'slurm':
        set_SLURM_vars(args)
    elif args.distrun == 'torchrun':
        set_TORCHRUN_vars(args)
    else:
        set_vars(args)

    print(args)

    set_deterministic()

    args.device = f'cuda:{args.local_rank}'
    torch.cuda.set_device(args.device)

    dist.init_process_group(backend='nccl', world_size=args.world_size, rank=args.ra
nk)

    # read sequences, create dataset
    S = Sequences(args.fname, args.mask_prob)

    # have to use distributed sampler for multiple GPUs
    sampler = DistributedSampler(
        S, rank=args.rank, num_replicas=args.world_size, shuffle=True
    )
    data_gen = data.DataLoader(S, batch_size=args.batch_size, sampler=sampler)

    # create the NN model
    model = BERT(
        args.d,
        args.dk,
        S.block_size,
        args.num_layers,
        args.num_heads,
        S.alphabet_idx,
        S.alphabet,
        args.use_sepheads,
    )
    model = model.to(args.device)

    # wrap model in DistributedDataParallel
    model = torch.nn.parallel.DistributedDataParallel(
        model, device_ids=[args.local_rank], output_device=args.local_rank
    )

    optimizer = optim.Adam(
        model.parameters(),
        lr=args.learning_rate,
        betas=(0.9, 0.98),
        weight_decay=args.weight_decay,
    )

    prev_e = 0
    if args.chkpt_fname is not None:
        # everyone resumes from checkpoint saved by rank=0 process
        saveinfo = torch.load(args.chkpt_fname)
        prev_e = saveinfo['epoch'] + 1
        model.load_state_dict(saveinfo['state_dict'])
        optimizer.load_state_dict(saveinfo['optimizer'])
        print("resume from epoch:", prev_e)  # resume from prev_e

    model.train()

    start_t = time.time()
    for e in range(prev_e, prev_e + args.epochs):
        sampler.set_epoch(e)  # set for distributed sampler
        running_loss = 0.0
```

global rank

0 . . . worldsize-1

```
        correct = 0.0
        num_masked = 0.0
        e_st = time.time()

        # only rank=0 print out info
        if args.rank == 0:
            totlen = len(S) // args.batch_size
            totlen = totlen // args.world_size
            enum_data = tqdm(enumerate(data_gen), total=totlen)
        else:
            enum_data = enumerate(data_gen)
        for bidx, (block, mask, labels) in enum_data:
            block = block.to(args.device)
            mask = mask.to(args.device)
            labels = labels.to(args.device)

            model.zero_grad()
            preds = model(block)

            # cross_entropy expects B x C x block_size
            preds = preds.permute(0, 2, 1)
            loss = nn.functional.cross_entropy(preds, labels, reduction='none')
            loss = torch.sum(loss * mask)

            loss.backward()
            optimizer.step()
            running_loss += loss.item()

            # compute number of correct predictions, keep track of num_masked for ac
c
            pred_labels = torch.argmax(preds, dim=1)
            correct += torch.sum((pred_labels == labels) * mask).item()
            num_masked += torch.sum(torch.where(mask == 1)[0]).item()

        e_en = time.time()
        if args.rank == 0:
            acc = correct / num_masked
            print(
                "epoch",
                e,
                running_loss,
                bidx,
                running_loss / (bidx + 1),
                acc,
                correct,
                num_masked,
                e_en - e_st,
            )
            model.module.checkpoint(args, bidx, e, running_loss, optimizer)

    # save the transformer model for downstream classification
    if args.rank == 0:
        model.module.save_transformer(args)

    torch.distributed.destroy_process_group()
    end_t = time.time()
    if args.rank == 0:
        print("finished in time", end_t - start_t, args.num_workers)
```