Bigram LLM Model

August 8, 2023

1 Bigram Model

The "Attention Is All You Need" paper introduced the revolutionary Transformer architecture, which has since become a cornerstone in modern NLP. This project aims to leverage the Transformer's attention mechanisms to build a bigram language model that predicts the next word in a sentence given the previous words.

[ ]:
```
# Checking CUDA (GPU)
!nvidia-smi
```

```
Fri Apr 14 10:29:18 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   60C    P8    11W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[ ]:
```
import torch
import torch.nn as nn
from torch.nn import functional as F
torch.cuda.is_available()
```

[ ]:
```
True
```

2 Preprocessing and Data Loading

[ ]:
```
# Download Data
!wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
filename = '/content/drive/MyDrive/Colab Notebooks/content/shakespeareInputLLM.txt'
with open(filename, 'r', encoding='utf-8') as f:
    text = f.read()
```

[ ]:
```
# Create vocabulary
torch.manual_seed(1337)

chars = sorted(list(set(text)))
vocab_size = len(chars)
# 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: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string
```

[ ]:
```
# Train and test splits
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:]
```

[ ]:
```
# Data loading
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])
    x, y = x.to(device), y.to(device) # sending data to device (cpu or gpu)
    return x, y
```

[ ]:
```
# Define hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 5000
learning_rate = 3e-4 #self-attention can't tolerate high lr
device = 'cuda' if torch.cuda.is_available() else 'cpu' # if you have gpu, run on it
eval_interval = 500
```

```
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
```

[ ]:
```
# Let's estimate the loss every few epochs
@torch.no_grad()
def estimate_loss():
    out = {}
    '''
        Instead of evaluating the model every iter,
        we Evaluate every eval_interval based on what the model has been␣
 ↪trained so far
    '''
    model.eval() # Switch to Evaluation mode
    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() # Switch to Training mode
    return out
```

3 Transformer Architecture

[ ]:
```
# Self-Attention
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.head_size = head_size
        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_size))) #tril is not a parameter to be optimized

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)   # (B, T, head_size)
```

```
        q = self.query(x) # (B, T, head_size)
        v = self.value(x) # (B, T, head_size)

        # compute attention scores ("affinities")
        wei = (q @ k.transpose(-2, -1)) * self.head_size**-0.5   # (B, T,␣
↪head_size) @ (B, head_size, T) -> (B, T, T)  # Notice it has changed to␣
↪(Block_size, BLock_size)
        wei = wei.masked_fill(self.tril[:T, :T]==0, float('-inf'))  # in lower␣
↪traiangular matrix filled with 1's on lower left and 0's on upper right, we␣
↪are replacing 0's with '-inf' # (B, T, T)
        wei = F.softmax(wei, dim=-1)  # (B, T, T)
        wei = self.dropout(wei)

        # perform weighted aggregation of values
        out = wei @ v  # (B, T, T) @ (B, T, head_size) -> (B, T, head_size) ==␣
↪(B, T, n_embd) if num_heads = 1
        return out

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) # (B, T,␣
↪head_size*4) == (B, T, n_embd)
        out = self.proj(out) # we are doing Linear transformation of the output␣
↪from self-attention
        out = self.dropout(out)
        return out

class FeedForward(nn.Module):
    """ Simple Linear Layer followed by Non-Linearity"""
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4* n_embd), # making ouput of Linear 4 times, as␣
↪suggested in paper
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd), # self.proj # here again making␣
↪ouput of Linear n_embd, as suggested in paper
            nn.Dropout(dropout)
```

```python
        )

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

class Block(nn.Module):
    """ Transformer Decoder Block: communication followed by computation """
    def __init__(self, n_embd, n_head):
        super().__init__()
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        # Self Attention
        head_size = n_embd // n_head
        self.sa_heads = MultiHeadAttention(n_head, head_size)
        # Feed Forward
        self.ffwd = FeedForward(n_embd)
        # Layer Norms
        self.ln1 = nn.LayerNorm(n_embd) # making Unit Gaussians
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa_heads(self.ln1(x)) # in original paper the layernorm is applied after the computation, though overtime it has become more common to apply it before the computaion
        x = x + self.ffwd(self.ln2(x))
        return x

class BigramLanguageModel(nn.Module):
    '''super simple bigram model - Decoder-only Transformer'''
    def __init__(self):
        super().__init__()
        # Each word or token will be represented by an n_embd-dimensional embedding vector.
        self.token_embedding_table = nn.Embedding(num_embeddings=vocab_size, embedding_dim=n_embd)
        # We don't just want to encode the identity of token, but also its position
        self.position_embedding_table = nn.Embedding(num_embeddings=block_size, embedding_dim=n_embd)
        # Decoder Block Component
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        # Layer Norm
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        # Applies a linear transformation to the incoming data: :math:`y = x @ W.T + b`
        self.lm_head = nn.Linear(in_features=n_embd, out_features=vocab_size)
```

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

        # idx and targets are both (B,T) tensor of integers
        tkn_emb = self.token_embedding_table(idx) # (B,T,C)

        # range(0, T):: Every position will have n_embd-dimensional embedding vector.
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)

        # x now, not just hold token identity but also its position
        x = tkn_emb + pos_emb # (B, T, C)

        # Block component
        x = self.blocks(x) # (B, T, C)

        # final layer norm
        x = self.ln_f(x) # (B, T, C)

        # Making Logits
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape # (B,T,vocab_size)
            logits = logits.view(B*T, C) # (B*T,vocab_size)
            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):
            # as we are implementing position embedding (pos_emb), we can't include context more than block_size
            # crop idx to the last block_size token
            idx_cond = idx[:, -block_size:] # (B, block_size) == (B, T)
            # get the predictions
            logits, _ = self(idx_cond) #forward()
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, 1, C) == (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, 1, C) == (B, C)
```

```python
            # 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

# Build Model
model = BigramLanguageModel()
m = model.to(device)

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

# Epochs
for iter in range(max_iters):

    # After every eval_interval iters, evaluate the loss on train and val sets
    if (iter % eval_interval == 0) or (iter==max_iters-1):
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss␣
 ↪{losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch('train') # (B, T)

    # train model
    logits, loss = model.forward(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

# Generate next word using the model
context = torch.zeros((1, 1), dtype=torch.long, device=device) # (B=1, T=1)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
```

```
step 0: train loss 4.2846, val loss 4.2820
step 500: train loss 1.8865, val loss 2.0023
step 1000: train loss 1.5361, val loss 1.7221
step 1500: train loss 1.3948, val loss 1.6038
step 2000: train loss 1.3077, val loss 1.5490
step 2500: train loss 1.2523, val loss 1.5153
step 3000: train loss 1.2010, val loss 1.4894
step 3500: train loss 1.1587, val loss 1.4800
step 4000: train loss 1.1222, val loss 1.4800
step 4500: train loss 1.0853, val loss 1.4736
step 4999: train loss 1.0494, val loss 1.4913
```

```
But with price of a breast sast-creature.
Of whom, Cariolanus: of God! what's Romeo?

Third Consciden:
Mistress, let's proceed. Go to me, go: I say.

CAMILLO:
By my lord, I'll braw a light:
I have bore alone death.

SLY:
If you would I wish vengeance me when I was off
these advancementary; this to fled till
I clear thee join till ar outrain, thou art to me;
And, not many hath punishmen.

SLY:
They Gentleman, I have deliver'd by thus leave
He known I dissevel to you!

Lord:
Lords, am I though for
```

[ ]:
```python
print("Total Number of Parameters:", sum(p.numel() for p in m.parameters()))
```

```
Total Number of Parameters: 10788929
```

[ ]:
```python
# Save the model
PATH = '/content/drive/MyDrive/Colab Notebooks/content/BigramModel.pth'
torch.save(m, PATH)
```

[ ]:
```python

```