class TransformerBlockPreNorm(Module):

    """ Single transformer block with attention plus a feedforward network"""

    def \_\_init\_\_(self, d,heads,d\_ff):

        super().\_\_init\_\_()

        self.attn=SelfAttention(d, heads)

        self.w1=Linear(d,d\_ff)

        self.w2=Linear(d\_ff,d)

        self.norm1=LayerNorm(d)

        self.norm2=LayerNorm(d)

        self.relu=ReLU()

    def forward(self,X,mask=None):

        Z=X+self.attn(self.norm1(X))

        Y=Z+self.w2(self.relu(self.w1(self.norm2(Z))))

        return Y

class LanguageModel(Module):

    """ Complete language model that runs:

        - Embedding

        - Positional Encoding

        - Tranformer block layers

        - Output linear layer

    """

    def \_\_init\_\_(self, hidden\_dim,seq\_len,heads,ffn\_dim,num\_layers,vocab\_size):

        super().\_\_init\_\_()

        self.hidden\_dim = hidden\_dim

        self.embed= Embedding(vocab\_size, hidden\_dim)

        self.position = PositionalEncoding(hidden\_dim)

        layerList=[]

        for i in range(num\_layers):

            layerList.append(TransformerBlockPreNorm(hidden\_dim,heads,ffn\_dim))

        self.layers=ModuleList(layerList)

        self.norm=LayerNorm(hidden\_dim)

        self.last\_layer=Linear(hidden\_dim, vocab\_size)

    def forward(self,X):

        B,T=X.shape[0],X.shape[1]

        X\_hidden=torch.zeros(B,T,self.hidden\_dim)

        for i in range(B):

            X\_hidden[i] = self.embed(X[i])

        X\_hidden = self.position(X\_hidden)

        mask=torch.tril(torch.ones(T,T))

        mask=mask.masked\_fill(mask==0,float('-inf'))

        for layer in self.layers:

            X\_hidden=layer(X\_hidden,mask)

        #print((self.norm(X\_hidden)).shape)

        return self.last\_layer(self.norm(X\_hidden))

def epoch\_transformer\_lm(model, tokens, seq\_len, opt=None, batch\_size=5, verbose=True):

    losses=[]

    big\_batch=seq\_len\*batch\_size

    num\_batches=tokens.shape[0]//big\_batch

    for i in range(num\_batches):

        #print(set.shape,num\_batches)

        t=tokens[i\*big\_batch:(i+1)\*big\_batch]

        t=t.reshape(batch\_size,seq\_len)

        X=t[:,0:seq\_len-1]

        targets=t[:,1:seq\_len]

        H=model(X)

        loss=F.cross\_entropy(H.reshape(-1,H.size(-1)),targets.reshape(-1))

        losses.append(loss)

        if opt is not None:

            loss.backward()

            opt.step()

            opt.zero\_grad()

        if i%10==0:

            print(f"Batch {i+1}/{num\_batches},Loss: {loss.item()}")

    return losses

embed\_dim=128

d\_ff=1024

totalLayers=5

attnHeads=8

model = LanguageModel(embed\_dim, seq\_len, attnHeads, d\_ff, totalLayers, tokenizer.vocab\_size)

#opt = optim.Adam(model.parameters(), lr=1e-3)

opt = SGD(model.parameters(), lr=0.5)

sum(p.numel() for p in model.parameters())

Output – 9892554

Case 1:

embed\_dim=256

d\_ff=1024

totalLayers=5

attnHeads=8

model = LanguageModel(embed\_dim, seq\_len, attnHeads, d\_ff, totalLayers, tokenizer.vocab\_size)

#opt = optim.Adam(model.parameters(), lr=1e-3)

opt = SGD(model.parameters(), lr=0.5)

A graph of training and test loci

Description automatically generated Average Test Loss= 6.01

Case 2:

embed\_dim=128

d\_ff=1024

totalLayers=5

attnHeads=8

model = LanguageModel(embed\_dim, seq\_len, attnHeads, d\_ff, totalLayers, tokenizer.vocab\_size)

#opt = optim.Adam(model.parameters(), lr=1e-3)

opt = SGD(model.parameters(), lr=0.5)

A graph of a graph with blue and orange lines

Description automatically generated Average Test Loss= 5.8

# KV Cache Implementation

class SelfAttention(Module):

    """ Multi-head self attention"""

    def \_\_init\_\_(self, d, num\_heads, max\_seq\_length=None):

        super().\_\_init\_\_()

        self.wq = Linear(d,d)

        self.wk = Linear(d,d)

        self.wv = Linear(d,d)

        self.wo = Linear(d,d)

        self.dim = d // num\_heads

        self.num\_heads = num\_heads

        self.max\_seq\_length=max\_seq\_length

        assert(self.dim \* num\_heads == d)

        self.i=0

        self.k\_cache=None

        self.v\_cache=None

    def clear\_cache(self):

        self.k\_cache=self.k\_cache[:,-self.max\_seq\_length:,:]

        self.v\_cache=self.v\_cache[:,-self.max\_seq\_length:,:]

    def forward(self, X, mask = None, use\_kv\_cache=False):

        # X in (B x T x d)

        B, T, d = X.shape

        # Q, K, V => (B x h x T x d/h)

        Q = self.wq(X).view(B, T, self.num\_heads, self.dim).transpose(1,2)

        K = self.wk(X).view(B, T, self.num\_heads, self.dim).transpose(1,2)

        V = self.wv(X).view(B, T, self.num\_heads, self.dim).transpose(1,2)

        if use\_kv\_cache and self.k\_cache is not None:

            K=torch.cat((self.k\_cache,K),dim=2)

            V=torch.cat((self.v\_cache,V),dim=2)

            if self.max\_seq\_length is not None:

                #print("KV Cache")

                self.clear\_cache()

            self.k\_cache=K

            self.v\_cache=V

        #Q @ K.T => B x h x T x T

        scores = Q @ K.transpose(2,3) / np.sqrt(self.dim)

        if mask is not None:

            scores += mask

        A = torch.softmax(scores, -1)

        return self.wo((A @ V).transpose(1,2).contiguous().view\_as(X))

def sample\_transformer\_naive(model, tokens, max\_context, num\_tokens, temperature=0.6,use\_kv\_cache=True):

    for i in range(num\_tokens):

        probs = torch.softmax(model(tokens[None,i:i+1],use\_kv\_cache)[0,-1] / temperature,-1)

        tokens = torch.cat([tokens, torch.multinomial(probs, 1)])

    return tokens

## Generated text:

of anyone anywhere in the United States and

most your the the the Prince; noble, When of he

good presence the

,

I my. I time

King What

To. love

do, To, a What,

.

.

.

king, and

For

Let

I,

death My For me

,

That. lord my, Enter and

That To world

, lord

to the The the?.

I. With the skill bargain truth

...

speak not,, Than have My To That For

my

.,