PA2 (Decoder Part)

dictionary with Configuration details

```
pprint(config)
```

```
{'input': {'batch_size': 10, 'embed_dim': 32, 'seq_len': 8, 'vocab_size': 12},
 'model': {'d_ff': 128,
           'd_model': 32,
           'dk': 4,
           'dq': 4,
           'dv': 4,
           'n_heads': 8,
           'n_layers': 6}}
```

```
vocab_size = config['input']['vocab_size']
batch_size = config['input']['batch_size']
seq_len = config['input']['seq_len']
embed_dim = config['input']['embed_dim']
dmodel = embed_dim
dq = torch.tensor(config['model']['dq'])
dk = torch.tensor(config['model']['dk'])
dv = torch.tensor(config['model']['dv'])
heads = torch.tensor(config['model']['n_heads'])
d_ff = config['model']['d_ff']
```

extracting parameters names from configuration dictionary

```
# do not edit this cell
data_url = 'https://github.com/Arunprakash-A/LLM-from-scratc
r = requests.get(data_url)
label_ids = torch.load(BytesIO(r.content),weights_only=True)
print(label_ids, label_ids.size())
```

```
tensor([[ 7,  8,  7,  7,  9,  2,  6],
        [10,  1, 10,  5,  3,  6,  8],
        [ 3,  4,  8,  2, 10, 10, 10],
        [ 4, 10,  1,  3,  4,  9,  7],
        [ 8,  4,  7,  3,  8, 10,  5],
        [ 9,  1,  8,  5,  9,  9, 10],
        [ 7,  3,  8,  2,  5,  1,  5],
        [ 3,  3,  2,  1,  4,  1,  1],
        [10,  9,  9,  9,  6,  9,  2],
        [ 3,  6,  6,  3,  5,  4,  5]]) torch.Size([10, 7])
```

getting label_ids.

10x7

batch size → 10

token_size → 7

seq_len is 8 (as per config)

We add 0 for <Go> to every sequence in starting

```
# Insert a special [start] token (ID = 0) at the beginning of label_ids
start_token = torch.zeros(label_ids.shape[0], 1, dtype=torch.long)
#print(start_token, start_token.shape)
token_ids = torch.cat((start_token, label_ids), dim=1)

print(token_ids, token_ids.size())
```

→ 0 0 0 0 0 0 0 0 0 0

```
tensor([[ 0,  7,  8,  7,  7,  9,  2,  6],
        [ 0, 10,  1, 10,  5,  3,  6,  8],
        [ 0,  3,  4,  8,  2, 10, 10, 10],
        [ 0,  4, 10,  1,  3,  4,  9,  7],
        [ 0,  8,  4,  7,  3,  8, 10,  5],
        [ 0,  9,  1,  8,  5,  9,  9, 10],
        [ 0,  7,  3,  8,  2,  5,  1,  5],
        [ 0,  3,  3,  2,  1,  4,  1,  1],
        [ 0, 10,  9,  9,  9,  6,  9,  2],
        [ 0,  3,  6,  6,  3,  5,  4,  5]]) torch.Size([10, 8])
```

Now we have 10 sequences of size 8.

Now we have to implement Masked attention, Cross attention and FFN.

```python
class MHMA(nn.Module):
    def __init__(self, dmodel, dq, dk, dv, heads, mask=None):
        super(MHMA, self).__init__()
        self.dmodel = dmodel
        self.dq = dq
        self.dk = dk
        self.dv = dv
        self.heads = heads
        # Initialize weights for Q, K, V, and output using random seeds
        torch.manual_seed(43)
        self.WQ = nn.Parameter(torch.randn(dq * heads, dmodel))
        torch.manual_seed(44)
        self.WK = nn.Parameter(torch.randn(dk * heads, dmodel))
        torch.manual_seed(45)
        self.WV = nn.Parameter(torch.randn(dv * heads, dmodel))
        torch.manual_seed(46)
        self.WO = nn.Parameter(torch.randn(dmodel, dv * heads))
        self.mask = mask
```

— Constructor

→ calls _init_ of parent class nn.Module.

Class parameters/attributes.

initialisation.
nn.Parameter( ) allows self.WQ to be updated during training

WQ, WK, WV, WO → 32×32

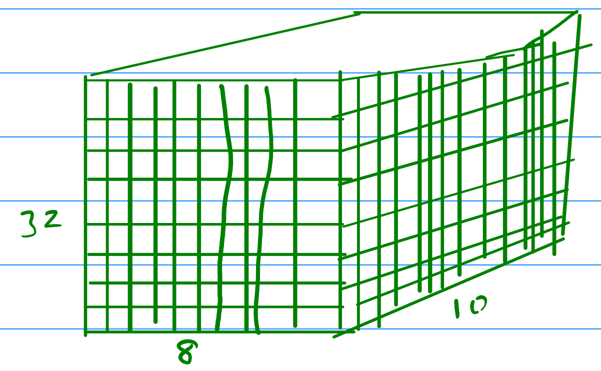
```python
    def forward(self, x):
        # Linear transformations for Q, K, V
        Q = torch.matmul(x, self.WQ.T)  # (batch_size, seq_len, dq * heads)
        K = torch.matmul(x, self.WK.T)  # (batch_size, seq_len, dk * heads)
        V = torch.matmul(x, self.WV.T)  # (batch_size, seq_len, dv * heads)
        # Reshape Q, K, V for multi-head computation
        batch_size = Q.shape[0]
        seq_len = Q.shape[1]
        Q = Q.view(batch_size, seq_len, self.heads, self.dq).transpose(1, 2)  # (batch_size, heads, seq_len, dq)
        K = K.view(batch_size, seq_len, self.heads, self.dk).transpose(1, 2)  # (batch_size, heads, seq_len, dk)
        V = V.view(batch_size, seq_len, self.heads, self.dv).transpose(1, 2)  # (batch_size, heads, seq_len, dv)
        # Scaled dot-product attention
        dk = torch.tensor(self.dk, dtype=torch.float32)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(dk)  # (batch_size, heads, seq_len, seq_len)
        # Apply mask (if mask not provided)
        if self.mask is None:
            self.mask = torch.triu(torch.ones((seq_len, seq_len)), diagonal=1).to(scores.device)
            self.mask = self.mask == 1  # Convert to boolean mask
        scores = scores.masked_fill(self.mask.unsqueeze(0).unsqueeze(0), float('-inf'))
        #print(scores) for debugging
        # Apply softmax to obtain attention weights
        attn_weights = F.softmax(scores, dim=-1)  # (batch_size, heads, seq_len, seq_len)
        # Compute output
        out = torch.matmul(attn_weights, V)  # (batch_size, heads, seq_len, dv)
        # Concatenate heads
        out = out.transpose(1, 2).contiguous().view(batch_size, seq_len, self.heads * self.dv)
        # Final linear transformation
        out = torch.matmul(out, self.WO.T)  # (batch_size, seq_len, dmodel)
        # print("Size of output from Mask Attention Layer is\n", out.size())for debugging
        return out
```

X → 10×8×32
WQ.T → 32×32
output will be of size
10×8×32
0 1 2

WQ, WK, WV and WO $\longrightarrow$ 32 × 32   BS seq_len, emb_dim

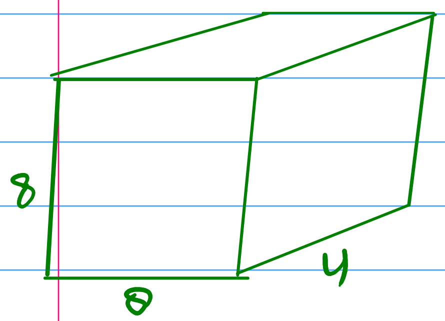
Q $\rightarrow$ matmul (X, WQ.T)   X size 10 × 8 × 32.

Same size for K and V.   ∴ Q size $\longrightarrow$ 10 × 8 × 32 → 3D Tensor (BS, seq_len, emb_dim.)



Q = Q.view(batch_size, seq_len, self.heads, self.dq)

$\rightarrow$ reshapes 10 × 8 × 32 tensor into 10 × 8 × 8 × 4 tensor → 4D tensor (BS, seq_len, heads, dq.)



$\longrightarrow$ 10 such tensors.

Q = Q. transpose (1,2) means

0	1	2	3
10 ×	8 ×	8 ×	4

swaps second and third dimension

So now Q $\rightarrow$ 10 × 8 × 8 × 4 (batch size., heads, seq_len, dq)

Similar operation for K and V.

Now Scores = matmul (Q, K.T(-2,-1)) / sqrt (dk)

Q $\rightarrow$ 10 × 8 × 8 × 4 (BatchSize, Heads, seq len, dQ.)

K $\rightarrow$ Same.

K.T (-2,-1) swaps last and second last dimension

∴ K.T (-2,-1) $\longrightarrow$ 10 × 8 × 4 × 8 (bs, heads, dk, seq_len) now marmul (10 × 8 × 8 × 4 , 10 × 8 × 4 × 8)

Batch seq head dq   bs head dk seq-len

dk and dq are same.

output will be 10 × 8 × $\boxed{8 \times 8}$

batchsize heads seq_len seq_len

So for every head, every batch we get Scores.

Specifically, for each head, `scores[i, h, j, k]` represents the dot product (similarity) between the `j`-th query vector in the `i`-th batch and the `k`-th key vector in that sequence for the `h`-th head.

Now Attention_weights = Softmax (scores, dim=-1)

10 × 8 × 8 × 8
bs heads seq_len seq_len.

out = marmul (Attention_weights, V)

10 × 8 × 8 × 8 , 10 × 8 × 8 × 4

= 10 × 8 × $\boxed{8 \times 4}$
bs head seq_len dv

matrix multiplication along last 2 dimension
8 × 8   8 × 4
→ 8 × 4

swap

```
# concatenate heads
out = out.transpose(1, 2).contiguous().view(batch_size, seq_len, self.heads * self.dv)
```

(concatenation)

10 × 8 × 8 × 4
bs seq_len head dv

Ensuring Contiguous Memory Layout:

- `.contiguous()` ensures that the tensor's memory layout is contiguous before reshaping. This is important for efficiency when reshaping a tensor.
- Without `.contiguous()`, the view operation might fail if the tensor's memory layout is not contiguous.

.View → reshape in 10 × 8 × 32

heads × dq   d_model

wo.T → 32 × 32.

Out = marmul (out , self.wo.T)

→ 10 × 8 × 32 original dimension

```
scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(dk)  # (batch_size, heads, seq_len, seq_len)
    # Apply mask (if mask not provided)
    if self.mask is None:
        self.mask = torch.triu(torch.ones((seq_len, seq_len)), diagonal=1).to(scores.device)
        self.mask = self.mask == 1  # Convert to boolean mask
    scores = scores.masked_fill(self.mask.unsqueeze(0).unsqueeze(0), float('-inf'))
    #print(scores) for debugging
    # Apply softmax to obtain attention weights
    attn_weights = F.softmax(scores, dim=-1)  # (batch_size, heads, seq_len, seq_len)
```

Scores → 10 × 8 × 8 × 8.
BS Head seq_len seq_len.

```
print(torch.triu(torch.ones((seq_len, seq_len)), diagonal=1))
```

```
tensor([[0., 1., 1., 1., 1., 1., 1., 1.],
        [0., 0., 1., 1., 1., 1., 1., 1.],
        [0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1.],
        [0., 0., 0., 0., 0., 0., 1., 1.],
        [0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```
8×8

`.to(scores.device)` ensures that the mask is placed on the same device as the `scores` tensor, which could be a CPU or GPU.

self.mask = self.mask == 1

```
tensor([[False,  True,  True,  True,  True,  True,  True,  True],
        [False, False,  True,  True,  True,  True,  True,  True],
        [False, False, False,  True,  True,  True,  True,  True],
        [False, False, False, False,  True,  True,  True,  True],
        [False, False, False, False, False,  True,  True,  True],
        [False, False, False, False, False, False,  True,  True],
        [False, False, False, False, False, False, False,  True],
        [False, False, False, False, False, False, False, False]])
```

```
scores = scores.masked_fill(self.mask.unsqueeze(0).unsqueeze(0), float('-inf'))
```

`masked_fill()` is used to modify the `scores` tensor wherever the mask is `True`.

mask size of 8 × 8.

this changes it to 1 × 1 × 8 × 8.

and now fills score with −∞ wherever mask → True

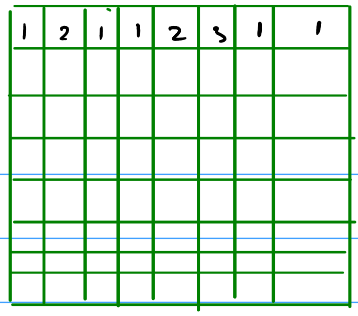
Scores → 10 × 8 × 8 × 8
bs heads seq_len seq_len

mask 1 × 1 × 8 × 8
seq_len seq_len

Scores

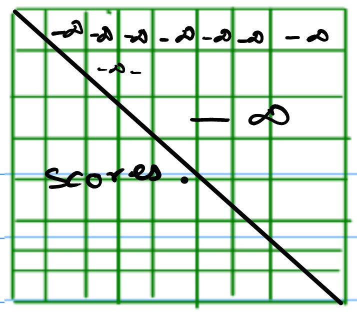
10×8×

8



8

mask →



MHCA./MHA Same

Same as MHMA

```python
class MHCA(nn.Module):
    def __init__(self, dmodel, dq, dk, dv, heads):
        super(MHCA, self).__init__()

        self.dmodel = dmodel
        self.dq = dq
        self.dk = dk
        self.dv = dv
        self.heads = heads

        # Initialize weights for Q, K, V, and output using the specified seeds
        torch.manual_seed(43)
        self.WQ = nn.Parameter(torch.randn(dq * heads, dmodel))

        torch.manual_seed(44)
        self.WK = nn.Parameter(torch.randn(dk * heads, dmodel))

        torch.manual_seed(45)
        self.WV = nn.Parameter(torch.randn(dv * heads, dmodel))

        torch.manual_seed(46)
        self.WO = nn.Parameter(torch.randn(dmodel, dv * heads))
```

→ from masked attention layer.

→ from encoder output.

```python
def forward(self, query, key, value):
    # Query is the output from the masked attention sub-layer
    # Key and Value come from the encoder output (which is fixed to a random matrix here)

    # Linear transformations for Q, K, V
    Q = torch.matmul(query, self.WQ.T)  # (batch_size, seq_len, dq * heads)
    K = torch.matmul(key, self.WK.T)    # (batch_size, seq_len, dk * heads)
    V = torch.matmul(value, self.WV.T)  # (batch_size, seq_len, dv * heads)

    # Reshape Q, K, V for multi-head computation
    batch_size = Q.shape[0]
    seq_len = Q.shape[1]

    Q = Q.view(batch_size, seq_len, self.heads, self.dq).transpose(1, 2)  # (batch_size, heads, seq_len, dq)
    K = K.view(batch_size, seq_len, self.heads, self.dk).transpose(1, 2)  # (batch_size, heads, seq_len, dk)
    V = V.view(batch_size, seq_len, self.heads, self.dv).transpose(1, 2)  # (batch_size, heads, seq_len, dv)

    # Scaled dot-product attention
    dk = torch.tensor(self.dk, dtype=torch.float32)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(dk)  # (batch_size, heads, seq_len, seq_len)
    attn_weights = F.softmax(scores, dim=-1)

    # Compute output
    out = torch.matmul(attn_weights, V)  # (batch_size, heads, seq_len, dv)

    # Concatenate heads
    out = out.transpose(1, 2).contiguous().view(batch_size, seq_len, self.heads * self.dv)

    # Final linear transformation
    out = torch.matmul(out, self.WO.T)  # (batch_size, seq_len, dmodel)
    #print("Size of output from Cross Attention Layer is\n", out.size())for debugging
    return out
```

```python
class FFN(nn.Module):
    def __init__(self, dmodel, d_ff):
        super(FFN, self).__init__()

        # First linear layer maps dmodel -> d_ff
        self.linear1 = nn.Linear(dmodel, d_ff)

        # Second linear layer maps d_ff -> dmodel
        self.linear2 = nn.Linear(d_ff, dmodel)

    def forward(self, x):
        # First linear layer followed by ReLU activation
        x = F.relu(self.linear1(x))

        # Second linear layer
        out = self.linear2(x)
        #print("Size of output from FFN layer is\n", out.size())for debugging
        return out
```

FFN layer.

nn.Linear ( ) has bias = True as default.

w_1 w_2

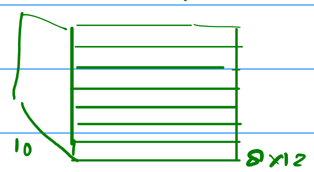
$32 \times 128 + 128$ $128 \times 32 + 32$.

```python
class OutputLayer(nn.Module):
    def __init__(self, dmodel, vocab_size):
        super(OutputLayer, self).__init__()

        # Linear layer mapping dmodel -> vocab_size to predict token IDs
        self.linear = nn.Linear(dmodel, vocab_size)

    def forward(self, x):
        # Apply linear layer to project to vocab size
        out = self.linear(x)
        #print("Size of output from Output Layer is\n", out.size())for debugging
        return out
```

Output layer.

$10 \times 8 \times 32$

$\rightarrow 10 \times 8 \times 12$




```python
class DecoderLayer(nn.Module):
    def __init__(self, dmodel, dq, dk, dv, d_ff, heads, mask=None):
        super(DecoderLayer, self).__init__()
        # Multi-head Masked Attention sub-layer
        self.mhma = MHMA(dmodel, dq, dk, dv, heads, mask=mask)
        # Multi-head Cross Attention sub-layer
        self.mhca = MHCA(dmodel, dq, dk, dv, heads)
        # Position-wise Feed Forward Network
        self.ffn = FFN(dmodel, d_ff)

        # Layer normalization for each sub-layer
        self.layer_norm_mhma = nn.LayerNorm(dmodel)
        self.layer_norm_mhca = nn.LayerNorm(dmodel)
        self.layer_norm_ffn = nn.LayerNorm(dmodel)

    def forward(self, x, encoder_output):
        # Multi-head Masked Self-Attention with residual connection and layer normalization
        mhma_output = self.mhma(x)
        x = self.layer_norm_mhma(x + mhma_output)

        # Multi-head Cross Attention with residual connection and layer normalization
        mhca_output = self.mhca(x, encoder_output, encoder_output)
        x = self.layer_norm_mhca(x + mhca_output)

        # Feed Forward Network with residual connection and layer normalization
        ffn_output = self.ffn(x)
        out = self.layer_norm_ffn(x + ffn_output)
        #print("Size of output from the Decoder layer is\n", out.size())for debugging
        return out
```

Decoder layer.

— mhma (MHMA object)

— mhca (MHCA object)

— ffn (FFN object)

→ normalisation for each sublayer.

residual

// ✓

// ✓

```python
class Embed(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        super(Embed, self).__init__()

        # Set the random seed for reproducibility
        torch.manual_seed(70)

        # Initialize the embedding layer
        self.embed = nn.Embedding(vocab_size, embed_dim)

    def forward(self, x):
        # Lookup the embeddings for the token_ids
        out = self.embed(x)
        #print("Size of output from Embedding Layer is\n", out.size())for debugging
        return out
```

Embed class to get embeddings of the input.

```python
class Decoder(nn.Module):
    def __init__(self, vocab_size, dmodel, dq, dk, dv, d_ff, heads, mask=None, num_layers=1):
        super(Decoder, self).__init__()

        # Embedding Layer for the target token IDs
        self.embed_lookup = Embed(vocab_size, dmodel)

        decoder_layer = DecoderLayer(dmodel, dq, dk, dv, d_ff, heads, mask)

        # Stack of decoder Layers
        self.dec_layers = nn.ModuleList([copy.deepcopy(decoder_layer) for _ in range(num_layers)])

        # Output Layer to project the decoder output to the vocabulary size
        self.output_layer = OutputLayer(dmodel, vocab_size)

    def forward(self, enc_rep, tar_token_ids):
        # Get embeddings for the target token IDs
        x = self.embed_lookup(tar_token_ids)

        # Pass through each decoder Layer
        for dec_layer in self.dec_layers:
            x = dec_layer(x, enc_rep)

        # Final output Layer to get the logits for the vocabulary
        out = self.output_layer(x)
        #print("Size of the final output from DECODER is\n", out.size())for debugging
        return out
```

Full decoder

- get embeddings
- pass through decoder layer.
- Get output from output layer.

```python
# do not edit this
enc_rep = torch.randn(size=(batch_size,seq_len,embed_dim),generator=torch.random.manual_seed(10))
print(enc_rep.size())
```

```
torch.Size([10, 8, 32])
```

Encoder output

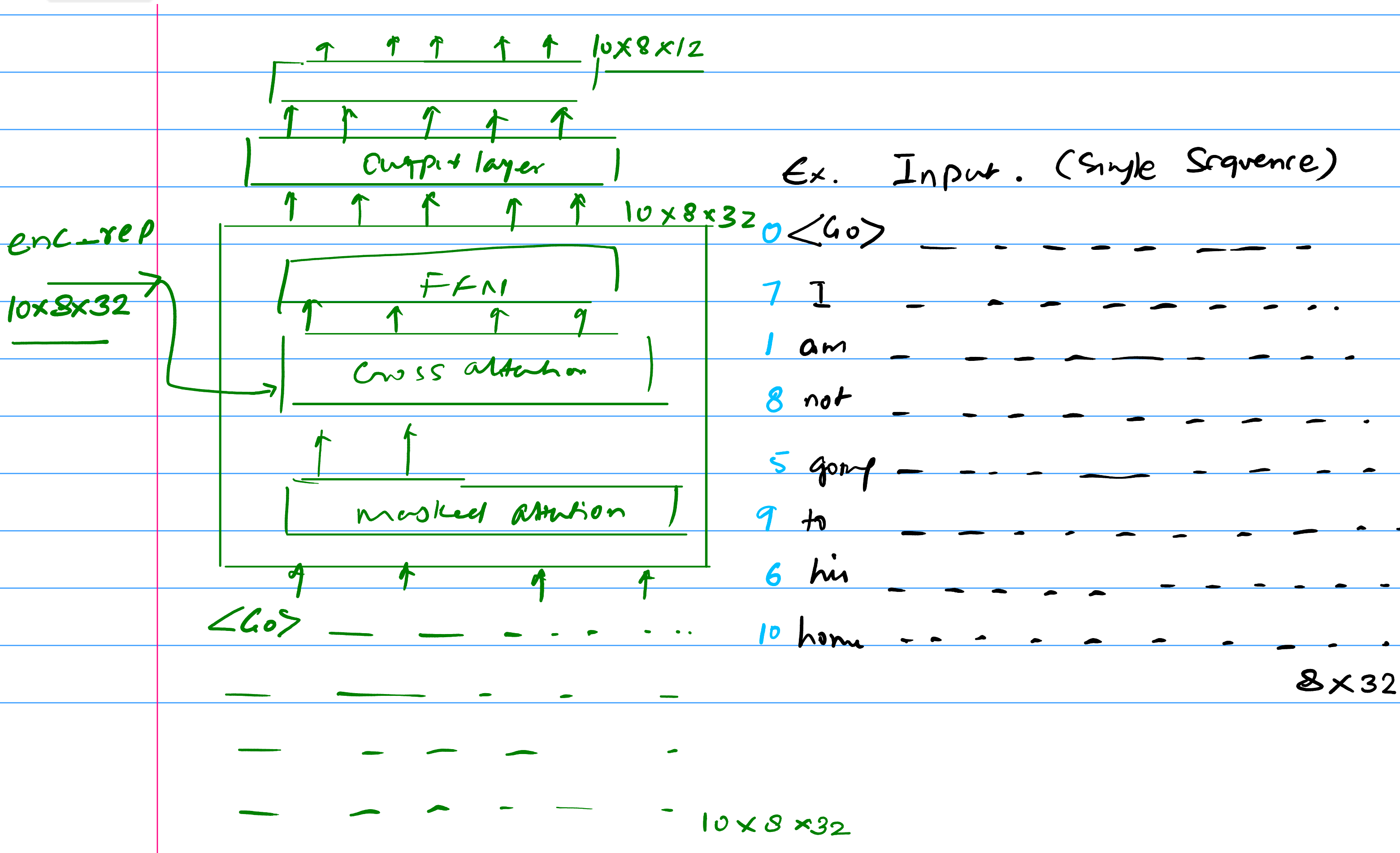
```python
model = Decoder(vocab_size,dmodel,dq,dk,dv,d_ff,heads,mask=None)
```

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```python
import matplotlib.pyplot as plt

def train(enc_rep, tar_token_ids, label_ids, epochs=1000):
    loss_trace = []
    for epoch in range(epochs):
        out = model(enc_rep, tar_token_ids)
        out = out.view(-1, vocab_size)
        target = tar_token_ids.view(-1)

        # Compute Loss
        loss = criterion(out, target.long())
        loss_trace.append(loss.item())  # Store the Loss value for visualization

        # Print Loss every 100 epochs
        if (epoch + 1) % 100 == 0:
            print(f'Loss in epoch - {epoch + 1} is {loss.item()}')

        # Backpropagation
        loss.backward()

        # Update parameters
        optimizer.step()
        optimizer.zero_grad()

    # Plot the Loss curve
    plt.plot(range(epochs), loss_trace, label='Training Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs. Epochs')
    plt.legend()
    plt.grid()
    plt.show()
```

`.view(-1, vocab_size)` changes the shape of `out` to have two dimensions:

- `-1` indicates that the size of this dimension should be inferred automatically.
- `vocab_size` is retained as the second dimension.



Output will be

Probability distribution.

$$\begin{bmatrix} 0.3 & 0.2 & 0.1 & 0.05 & 0.05 & 0.05 & \_ & \_ & \_ & \_ & \_ \\ \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \\ & & & & & & & & & & \end{bmatrix}_{8\times 12} \quad \Sigma = 1.$$

So Input $[\quad\quad]$ $[\quad\quad]$ $\dfrac{8\times12}{\text{output}}$

find Cross EntropyLoss → how many correctly identified.

Example

Token	0	6	1	8	4	10	7	5
Input	<Go>	I	am	not	going	to	my	home.

8×32^T

Target	I	am	not	going	to	my	home	<end>
	6	1	8	4	10	7	5	12

8×12^T

Input Tensor (enc_rep):

```
[   [0.1, 0.2, ..., 0.32],  # Start token
    [0.3, 0.4, ..., 0.32],  # "I"
    [0.5, 0.6, ..., 0.32],  # "am"
    [0.7, 0.8, ..., 0.32],  # "not"
    [0.9, 1.0, ..., 0.32],  # "going"
    [1.1, 1.2, ..., 0.32],  # "to"
    [1.3, 1.4, ..., 0.32],  # "my"
    [1.5, 1.6, ..., 0.32]   # "home"
]
```

Output Tensor (out):

```
[
    [0.1, 0.3, 0.2, ..., 0.1],  # Predictions for start token
    [0.4, 0.1, 0.2, ..., 0.3],  # Predictions for "I"
    [0.3, 0.5, 0.1, ..., 0.1],  # Predictions for "am"
    ...
    [0.2, 0.1, 0.4, ..., 0.2]   # Predictions for "home"
]
```

After reshaping out.view(-1, vocab_size)

```
[
    [0.1, 0.3, 0.2, ..., 0.1],  # Predictions for start token
    [0.4, 0.1, 0.2, ..., 0.3],  # Predictions for "I"
    [0.3, 0.5, 0.1, ..., 0.1],  # Predictions for "am"
    ...
    [0.2, 0.1, 0.4, ..., 0.2]   # Predictions for "home"
]
```

target = [0, 1, 2, ..., 7] # Example token IDs for each of the 8 tokens

Now Criterion (Cross Entropy Loss)

Full Example

The target token IDs are [0, 1, 2, 3, 4, 5, 6, 7], indicating the correct class for each token.

```
out = [
    [0.1, 0.3, 0.2, 0.1, 0.1, 0.05, 0.03, 0.01, 0.02, 0.01, 0.02, 0.02],  # Predictions for start token (ID 0)
    [0.4, 0.1, 0.2, 0.1, 0.05, 0.03, 0.01, 0.02, 0.01, 0.02, 0.01, 0.01],  # Predictions for "I" (ID 1)
    [0.3, 0.5, 0.1, 0.05, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01],  # Predictions for "am" (ID 2)
    [0.2, 0.3, 0.4, 0.1, 0.05, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01],  # Predictions for "not" (ID 3)
    [0.2, 0.1, 0.4, 0.1, 0.1, 0.05, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01],  # Predictions for "going" (ID 4)
    [0.2, 0.3, 0.1, 0.2, 0.1, 0.05, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01],  # Predictions for "to" (ID 5)
    [0.2, 0.1, 0.4, 0.1, 0.1, 0.05, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01],  # Predictions for "my" (ID 6)
    [0.1, 0.1, 0.4, 0.2, 0.1, 0.05, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01]  # Predictions for "home" (ID 7)
]
```

Cross-Entropy Loss Calculation

Mathematical Formula

The formula for the cross-entropy loss L for a single sample is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log(p(y_i))$$

where:

N is the number of classes (in this case, 12 for the vocabulary size).

y_i is the target class (the correct token ID).

$p(y_i)$ is the predicted probability of the correct class.

Step-by-Step Calculation

1. Identify the Target Classes and Corresponding Probabilities:
   - For each position t in the sequence, find the predicted probability corresponding to the target class.

Token Position t	Target Token ID y_t	Predicted Probabilities $p(y_t)$
0	0	0.1
1	1	0.1
2	2	0.1
3	3	0.1
4	4	0.4
5	5	0.1
6	6	0.02
7	7	0.01

$$\therefore \text{loss} = -\frac{1}{8}\left(5 \times \ln(0.1) + \ln(0.4) + \ln(0.01) + \ln(0.02)\right)$$

```
Parameter name: embed_lookup.embed.weight, Number of parameters: 384
Parameter name: dec_layers.0.mhma.WQ, Number of parameters: 1024
Parameter name: dec_layers.0.mhma.WK, Number of parameters: 1024
Parameter name: dec_layers.0.mhma.WV, Number of parameters: 1024
Parameter name: dec_layers.0.mhma.WO, Number of parameters: 1024
Parameter name: dec_layers.0.mhca.WQ, Number of parameters: 1024
Parameter name: dec_layers.0.mhca.WK, Number of parameters: 1024
Parameter name: dec_layers.0.mhca.WV, Number of parameters: 1024
Parameter name: dec_layers.0.mhca.WO, Number of parameters: 1024
Parameter name: dec_layers.0.ffn.linear1.weight, Number of parameters: 4096
Parameter name: dec_layers.0.ffn.linear1.bias, Number of parameters: 128
Parameter name: dec_layers.0.ffn.linear2.weight, Number of parameters: 4096
Parameter name: dec_layers.0.ffn.linear2.bias, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_mhma.weight, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_mhma.bias, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_mhca.weight, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_mhca.bias, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_ffn.weight, Number of parameters: 32
Parameter name: dec_layers.0.layer_norm_ffn.bias, Number of parameters: 32
Parameter name: output_layer.linear.weight, Number of parameters: 384
Parameter name: output_layer.linear.bias, Number of parameters: 12
Total number of parameters in the model, including the embedding layer, is: 17516
```

$\rightarrow 12 \times 32 = 384$

$W_Q, W_K, W_V, W_O \rightarrow$ MHMA. $4 \times 32 \times 32$.

MHCA. $4 \times 32 \times 32$.

$W_1 \rightarrow 32 \times 128$

$b_1 \rightarrow 128$

$W_2 \rightarrow 128 \times 32$

$b_2 \rightarrow 32$

mhma normalisation layer wt and bias

MHCA norm layer wt & bias

FFN norm layer wt & bias

output layer. $32 \times 12 = 384$

bias $\rightarrow 12$.

Total P in the decoder.