1) **Embeddings**

Problem: Fomat Input Embeddings. Implement the embedding lookup, including the addition of positional encodings.

Algorithm: We are given as input a tensor of B sequences where each sequence has T integers. Each integer is a token which represents the index of a word in a tensor of integer word tokens. The input tensor (idx) is used to to get the integer word tokens associated with their indicies from the input tensor. We then create a 1-D tensor of the T position indicies. Tranformers do not have a sense of position of the word tokens in the sequence they're in so, positional embeddings are used to have this positional information of word tokens. The 1-D tensor of position indicies is used to get the tensor of integer tokens which represent the positional information of their respective word tokens. To format the input embeddings similiar to BERT, we concatenate the word token embeddings with their position embeddings.

2) **Multi-Head Self-Attention**

Problem: Implement multi-head self-attention

Purpose of Self-Attention: Suppose we want to translate this sentence: "The cat did not want to eat because it was full". It is clear to us that "it" refers to "cat" but that is not so to an algorithm. Self-attention allows a model to associate "it" with "cat" when processing the word "it".

Algorithm: Each word is embedded into a vector of size n_embd. We process a 3D-tensor of word tokens x. In tensor x, there are B sequences where each sequence has T word tokens and the size of the embedding vector representation for each word token is n_embd. For each word token we, create a query vector, key vector, and value vector by performing a linear transformation to tensor x. That is, we multiply each word token embedding vector by the weight query matrix, weight key matrix, and weight value matrix to get each word tokens respective query vector, key vector, and value vector. These new vectors have dimensionality of the embedding dimension divided by the number of heads. For each word token, we perform dot products between their query vectors and all key vectors to get a similiarity score between the current word token and each key. We then divide them by the square root of the dimension of the key vectors. Masked self-attention is performed where the future similarity scores are zeroed out so that the model doesn't take into account future words, only current and previous. Keys and values are causal. We pass the result to a softmax function to normalize the scores and make them positive and sum to one. We then summate the value vectors multiplied by the similiarity scores. After performing these computations for each word token, each word token gets a resulting z vector which holds contextual information of the word token. This resulting set of z vectors is fed to the next block in the transformer.

3) **Putting it all together**

Problem: Embed inputs, apply each transformer layer sequentially, and get logits for each possible output word using a classification layer.

Algorithm: We start by embedding word tokens into vectors using input tensor idx to identify the indicies of the word tokens. If the hidden cache tensor has outputs from the previous transformer layer(s) (i.e. hidden states), concatenate those outputs with the word embeddings. We pass the embeddings through the transformer blocks and apply a layer norm to speed up training and improving optimality with a stable convergence to a solution. The linear layer is responsible for getting words from token vectors since it is a neural network which projects those vectors to a logits vector which holds scores associated to words next in sequence. Those scores are fed into a softmax function which converts them to pobabilities. The logit with the highest probability is chosen and its associated word is outputed as next in sequence.

4) **Implement an Encoder Transformer**

Problem: Sentences come in different lengths so, pad tokens are needed to make the shorter sentences as long as the longest sentence. Remove pad tokens from attention mask so that they do not affect results.

Algorithm: Initialize the attention mask by creating a 1D tensor of integers from range 0 to num_tokens-1, inclusive. num_tokens is a tensor of shape batch_size which contains the length of each sequence. Reshape num_tokens to have shape (B, 1, 1) so that it becomes a 3D tensor that is comparable to attention_mask. Turn attention_mask into a boolean tensor. For each element in attention_mask, mark positions within the valid sequence length as 1 and paddded positions as 0.

5) **Implement an Decoder Transformer**

Problem: Modify the full attention mask to create a causal mask.

Algorithm: Create a lower trianglular matrix.

6) **Use your model to generate!**

Problem: Sample from your model max_new_tokens times. You should feed the predictions back into the model each time. Adjust the probability distribution to be more or less greedy using the temperature parameter

Algorithm: To sample from model, we iterate max_new_token times. Get logits from model using the index tensor. We specifically get the logits at the last time step for each sequence. Logits has shape (B, T) so, we use slicing notion to get the last element along the T dimension which holds the sequence length. Divide the logits by temperature to scale the logits. Apply softmax function to the scaled logits. Sample from multivariate distribution to get the next word token. Append the next word tokens to the current bed of tokens.

7) **Implement an Encoder Decoder Transformer**

Problem: Create an Encoder Decoder Model by combining your previous transformers. The Encoder should encode the tokens from prefix into an embeddings. Use these in the hidden_cache to condition decoder generation.

Algorithm: Get outputs from previous layer by encoding the input prefix. Use hidden states to get and indices tensor to get scores and their corresponding losses.

8) **prefix_generation function**

Problem: Adjust original generation function to work Encoder-Decoder models

Algorithm: Add prefix as a parameter to model along with the indices tensor.

9) **Training a Language Model from Scratch**

Problem: Modify the tokenizer, architecture, or hyperparameters to decrease loss and drive accuracy above 80%.

Algorithm:

To modify the hyperparameters to decrease loss and drive accuracy above 80% we

1) decrease the learning rate from 5e-4 to 3e-4
    a) increasing the step size in a transformer model causes it to diverge when training thus causing destabilization. Citation: https://arxiv.org/pdf/2303.06296

b) This was derived from transformer paper https://arxiv.org/pdf/1706.03762 as follows

$$lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

from paper

$step\_num$ = 100,000

$warmup\_steps$ = 4000

d_{model} = 128

$$lrate = 128^{-0.5} \cdot min(100,000^{-0.5}, 100,000 \cdot 4000^{-1.5})$$

$$= 0.000279508497$$

$$\approx 0.0003$$

$$= 3e\text{-}4$$

2) increase the batch size from 16 to 32
    a) chosen batch size is a power of two. Citation: https://ai.stackexchange.com/questions/9739/what-is-batch-batch-size-in-neural-networks
    b) when gradient descent is performed to minimize the loss, the larger batch size allows for averaging over more samples which stabilizes training and improves estimates
3) increase the maximum number of iterations from 10000 to 25000
    a) more iterations allows for more learning which improves model
4) increase the number of decoder layers from 4 to 6
    a) 6 decoder layers were chose like in the transformer paper: https://arxiv.org/pdf/1706.03762
5) increase the embedding dimension from 64 to 128
    a) embedding dimension of 128 chosen like in paper: https://arxiv.org/pdf/2004.02984
6) increase the number of attention heads from 2 to 8
    a) 8 attention heads were chosen in transformer paper: https://arxiv.org/pdf/1706.03762

PyTorch Citations:

torch.nn.Embedding(num_embeddings, embedding_dim)
Parameters:
num_embdeddings (int) - Size of the dictionary of embeddings
embedding_dim (int) - Size of each embedding vector
Input: List of indices
Output: Corresponding word embeddings
Usage: Used to store word embeddings and retrieve them using indicies. Embeddings are of a fixed dictionary and size.
Citation: https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html

torch.arange() → Tensor
Usage: returns a 1-D tensor with values from interval [start, end).
Ex:
```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
```
Citation: https://pytorch.org/docs/stable/generated/torch.arange.html

Tensor.expand(*sizes) → Tensor
Parameters:
*sizes (torch.Size or int...) – the desired expanded size
Usage: Returns a new view of the self tensor with singleton dimensions expanded to a larger size.
Ex:
```
>>> x = torch.tensor([[1], [2], [3]])
>>> x.size()
torch.Size([3, 1])
>>> x.expand(3, 4)
tensor([[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3]])
```
Citation:
https://pytorch.org/docs/stable/generated/torch.Tensor.expand.html#torch.Tensor.expand

torch.unsqueeze(input, dim) → Tensor
Parameters:
input (Tensor) – the input tensor
dim (int) – the index at which to insert the singleton dimension
Usage: Returns a new tensor with a dimension of size one inserted at the specified position.
Ex:
```
>>> x = torch.tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
tensor([[ 1,  2,  3,  4]])
>>> torch.unsqueeze(x, 1)
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```
Citation: https://pytorch.org/docs/stable/generated/torch.unsqueeze.html

torch.nn.Linear()
Usage: Apply linear transformation y = x * transpose(A) + b to input data x
Ex:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

Citation: https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

torch.nn.Dropout(p)
Usage: During training, randomly zero out some of the elements of the input tensor with probability p.
This is used to reduce overfitting in a model.
Citation: https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

torch.matmul()
Usage: matrix product of two tensors
Citation: https://pytorch.org/docs/stable/generated/torch.matmul.html

torch.sqrt(input, *, out=None) → Tensor
Usage: Returns a new tensor with the square-root of the elements of input
Citation: https://pytorch.org/docs/stable/generated/torch.sqrt.html

torch.reshape(input, shape) → Tensor
Parameters:
input (tensor) - the tensor to be reshaped
shape (tuple of integers) - the new shape
Usage: Returns a tensor with the same data and number of elements as input, but with the specified new shape.
Ex:

```
>>> a = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> print(a)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> print(a.reshape([4, 2]))
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
```

Citation: https://pytorch.org/docs/stable/generated/torch.reshape.html

torch.transpose(input, dim0, dim1) → Tensor
Parameters:
input (tensor) - input tensor
dim1 (int) - first dimension to be transposed
dim2 (int) - second dimension to be transposed
Usage: Returns a tensor that is a transposed version of the input. The given dimensions dim0 and dim1 are swapped.

Ex:

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 1.0028, -0.9893,  0.5809],
        [-0.1669,  0.7299,  0.4942]])
>>> torch.transpose(x, 0, 1)
tensor([[ 1.0028, -0.1669],
        [-0.9893,  0.7299],
        [ 0.5809,  0.4942]])
```

Citation: https://pytorch.org/docs/stable/generated/torch.transpose.html

Tensor.masked_fill(mask, value) → Tensor
Parameters:
mask (BoolTensor) – the boolean mask
value (float) – the value to fill in with
Usage: Fills elements of a tensor with a specified value where mask is true. Creates a new tensor without modifying the original.
Citation: https://pytorch.org/docs/stable/generated/torch.Tensor.masked_fill.html

torch.nn.functional.softmax(input, dim=None)
Parameters:
input (Tensor) – input
dim (int) – A dimension along which softmax will be computed.
Usage:
Apply a softmax function. Softmax is defined as:

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_i)}$$

Citation: https://pytorch.org/docs/stable/generated/torch.nn.functional.softmax.html

torch.nn.ModuleDict(modules=None)
Parameters:
modules (iterable, optional) – a mapping (dictionary) of (string: module) or an iterable of key-value pairs of type (string, module)
Usage: Holds submodules in a dictionary. Can be indexed like a regular Python dictionary.
Citation: https://pytorch.org/docs/stable/generated/torch.nn.ModuleDict.html

torch.cat(tensors, dim=0) → Tensor
Parameters:
tensors (sequence of Tensors) – any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.
dim (int, optional) – the dimension over which the tensors are concatenated
Usage: Concatenates the given sequence of seq tensors in the given dimension.
Citation: https://pytorch.org/docs/stable/generated/torch.cat.html

torch.nn.LayerNorm(normalized_shape)
Parameters:
normalized_shape (int or list or torch.Size) – input shape from an expected input

Usage: Applies Layer Normalization over a mini-batch of inputs.
Citation: https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html

torch.tril(input) → Tensor
Parameters:
input (Tensor) – the input tensor.
Usage: Returns the lower triangular part of the matrix (2-D tensor) or batch of matrices input, the other elements of the result tensor out are set to 0.
Ex:

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[-1.0813, -0.8619,  0.7105],
        [ 0.0935,  0.1380,  2.2112],
        [-0.3409, -0.9828,  0.0289]])
>>> torch.tril(a)
tensor([[-1.0813,  0.0000,  0.0000],
        [ 0.0935,  0.1380,  0.0000],
        [-0.3409, -0.9828,  0.0289]])
```

Citation: https://pytorch.org/docs/stable/generated/torch.tril.html

torch.zeros(*size) → Tensor
Parameters
size (int...) – a sequence of integers defining the shape of the output tensor.
Usage: Returns a tensor filled with the scalar value 0, with the shape defined by the variable argument size.
Ex:

```
>>> torch.zeros(2, 3)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

>>> torch.zeros(5)
tensor([ 0.,  0.,  0.,  0.,  0.])
```

Citation: https://pytorch.org/docs/stable/generated/torch.zeros.html

torch.multinomial(input, num_samples) → LongTensor
Parameters:
input (Tensor) – the input tensor containing probabilities
num_samples (int) – number of samples to draw
Usage: Returns a tensor where each row contains num_samples indices sampled from the multivariate probability distribution located in the corresponding row of tensor input.
Citation: https://pytorch.org/docs/stable/generated/torch.multinomial.html