Cheat Sheet: Generative AI-Language Modeling with Transformers

Package/ Method	Description	Code Example
Dataset()	This code loads the IMDB data set and initializes iterators for both the training and validation sets. It then creates an iterator data_itr from the training iterator train_iter and retrieves the next data sample using the next() function.	`# Load the data set` `train_iter, val_iter= IMDB()` `data_itr=iter(train_iter)` `next(data_itr)`
Text Pipeline ()	You can utilize PyTorch's torchtext library to streamline the text processing pipeline for NLP tasks. Specifically, get_tokenizer("basic_english") from torchtext.data.utils is used for tokenizing the text data into a list of tokens. This tokenization process is essential for converting raw text into a format that your model can interpret. Furthermore, build_vocab_from_iterator, another utility from torchtext.vocab, is leveraged to construct the vocabulary from the tokenized text. This function iterates through the tokenized data, capturing the unique tokens and associating them with indices, including special symbols like <unk> for unknown tokens, <pad> for padding, and <eos> for end-of-sentence markers. By specifying specials and	`# Define special symbols and indices` `UNK_IDX, PAD_IDX, EOS_IDX = 0, 1, 2` `# Make sure the tokens are in order of their indices to properly insert them in vocab` `special_symbols = ['<unk>', '<pad>', '<eos>']` `tokenizer = get_tokenizer("basic_english")` `def yield_tokens(data_iter):` `    for _, data_sample in data_iter:` `        yield tokenizer(data_sample) + ['<eos>']` `vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=special_symbols, special_first=True)` `vocab.set_default_index(UNK_IDX)` `text_to_index = lambda text: [vocab(token) for token in tokenizer(text)] + [EOS_IDX]` `index_to_text = lambda seq_en: " ".join([vocab.get_itos()[index] for index in seq_en if index != EOS_IDX])`

	special_first=True, you ensure these special tokens are prioritized and properly indexed in your vocabulary, setting the groundwork for effective model training.	
Creating data for next token prediction()	This code snippet demonstrates a critical step in preparing data for training a language model: generating input-target pairs, where each pair is used for the next token prediction. The function get_sample takes two parameters: block_size, which defines the maximum length of the text sample, and text, the input text from which the sample is generated. To create a diverse training set, torch.randint is used to select a random starting point within the text. This randomness ensures that the model encounters different segments of the text during training, which is vital for learning a robust representation of the language. The selected text segment (src_sequence) and its immediate next token (tgt_sequence) form a pair used to train the model to predict the next token given a sequence of tokens.	`def get_sample(block_size, text):` `# Determine the length of the input text` `sample_leg = len(text)` `# Calculate the stopping point for randomly selecting a sample` `# This ensures the selected sample doesn't exceed the text length` `random_sample_stop = sample_leg - block_size` `# Check if a random sample can be taken (if the text is longer than block_size)` `if random_sample_stop >= 1:` `# Randomly select a starting point for the sample` `random_start = torch.randint(low=0, high=random_sample_stop, size=(1,)).item()` `# Define the endpoint of the sample` `stop = random_start + block_size` `# Create the input and target sequences` `src_sequence = text[random_start:stop]` `tgt_sequence= text[random_start + 1:stop + 1]` `# Handle the case where the text length is exactly equal or less the block size` `else:` `# Start from the beginning and use the entire text` `random_start = 0` `stop = sample_leg` `src_sequence= text[random_start:stop]` `tgt_sequence = text[random_start + 1:stop]`

<table>
<tr>
<td></td>
<td></td>
<td><pre><code># Append an empty string to maintain sequence alignment
tgt_sequence.append('<|endoftext|>')

return src_sequence, tgt_sequence</code></pre></td>
</tr>
<tr>
<td>(Src, Tgt) pairs ()</td>
<td>This code snippet generates a sample pair for training a language model, where block_size determines the maximum length of the sample and text represents the input text. It then prints the source sequence (src_sequences) and the corresponding target sequence (tgt_sequence). The function get_sample randomly selects a segment of the text of length block_size, ensuring proper alignment between the source and target sequences. If the text is shorter than block_size, it uses the entire text.</td>
<td><pre><code>block_size=10
src_sequences, tgt_sequence=get_sample(block_size,
text)
print(src_sequences)
print(tgt_sequence)</code></pre></td>
</tr>
<tr>
<td>Collate function()</td>
<td>This code defines a function collate_batch to prepare batches of input-source and target sequences for training a language model. It iterates over a batch of data, generates source and target sequences using the get_sample function with a specified block size (BLOCK_SIZE), tokenizes the text using a tokenizer, and converts tokens to indices using a vocabulary. The sequences are then converted to PyTorch tensors and appended to the respective source and target batches. Finally, the</td>
<td><pre><code>BLOCK_SIZE=30
def collate_batch(batch):
src_batch, tgt_batch = [], []
DEVICE = torch.device("cuda"
if torch.cuda.is_available() else "cpu")
for _,_textt in batch:
src_sequence,tgt_sequence=get_sample(BLOCK_SIZE,tokenize
r(_textt))
src_sequence=vocab(src_sequence)
tgt_sequence=vocab(tgt_sequence)
src_sequence= torch.tensor(src_sequence,
dtype=torch.int64)
tgt_sequence = torch.tensor(tgt_sequence,
dtype=torch.int64)</code></pre></td>
</tr>
</table>

	function pads sequences within the batch to ensure uniform length and returns the processed batches.	`src_batch.append(src_sequence)` `tgt_batch.append(tgt_sequence)` `src_batch = pad_sequence(src_batch, padding_value=PAD_IDX, batch_first=False)` `tgt_batch = pad_sequence(tgt_batch, padding_value=PAD_IDX, batch_first=False)` `return src_batch.to(DEVICE), tgt_batch.to(DEVICE)` `BATCH_SIZE=1` `dataloader = DataLoader(train_iter, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch)` `val_dataloader= DataLoader(val_iter, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_batch)`
Masking future tokens: Causal mask()	These functions create masks used in transformer-based models for self-attention: generate_square_subsequent_mask (sz, device=DEVICE): Generates a mask to prevent attending to subsequent positions in a sequence during self-attention. create_mask(src, device=DEVICE): Creates a mask for the source sequence to ensure that each position can only attend to previous positions during self-attention.	`def generate_square_subsequent_mask(sz,device=DEVICE):` `mask = (torch.triu(torch.ones((sz, sz), device=device)) == 1).transpose(0, 1)` `mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))` `return mask` `def create_mask(src,device=DEVICE):` `src_seq_len = src.shape[0]` `src_mask= nn.Transformer.generate_square_subsequent_mask(None,src_seq_len).to(device)` `return src_mask`

Padding mask()	The code generates a boolean mask, src_padding_mask, indicating the presence of padding tokens (True) in the source sequence src. Each True value corresponds to a padding token, while False represents non-padding tokens. The mask is structured to align with the sequence dimensions for proper masking.	`src_padding_mask = (src == PAD_IDX).transpose(0, 1)` `src.T:tensor([[ 1, 1, 1, 1, 6, 169, 438, 709..` `padding_mask:tensor([[ True, True, True, True, False,` `False, False, False, ..`
Custom GPT model architecture()	This forward pass function applies positional embeddings to input embeddings, incorporates source masks if provided, and passes the input through a transformer encoder. Finally, it passes the output through a linear layer (lm_head).	`def forward(self,x,src_mask=None,key_padding_mask=None):` `# Add positional embeddings to the input embeddings` `x = self.embed(x)* math.sqrt(self.embed_size)` `x = self.positional_encoding(x)` `if src_mask is None:` `src_mask, src_padding_mask = create_mask(x)` `output = self.transformer_encoder(x, src_mask,` `key_padding_mask)` `x = self.lm_head(x)` `return x`
Creating a small instance of the model	This code snippet initializes a custom GPT (Generative Pre-trained Transformer) model with specified parameters such as embedding size, number of transformer encoder layers, number of attention heads, vocabulary size, and dropout probability. The model is then moved to the specified device (DEVICE).	`ntokens = len(vocab) # size of vocabulary` `emsize = 200 # embedding dimension` `nlayers = 2 # number of ``nn.TransformerEncoderLayer`` `in ``nn.TransformerEncoder`` `nhead = 2 # number of heads in ``nn.MultiheadAttention`` `dropout = 0.2 # dropout probability` `model = CustomGPTModel(embed_size=emsize,` `num_heads=nhead, num_layers=nlayers, vocab_size =` `ntokens,dropout=dropout).to(DEVICE)`

Calculate the loss	During loss calculation, the encoder model generates a source and a target. During prediction, the decoder model generates logits Class 1 and Class 2.	`src= srctgt[0]` `tgt= srctgt[1]` `logits = model(src)`
Loss_fn	In preparation for loss calculation, you can see the reshaping of logits, where each row corresponds to the prediction for a token, spanning across both the sequence and the batch dimensions. You can reshape the target tensor so that its elements correspond correctly to the logits. This process ensures that every row from the logits aligns with the appropriate target outcomes for accurate loss estimation.	`logits_flat = logits.reshape(-1, logits.shape[-1])` `loss = loss_fn(logits_flat, tgt.reshape(-1))`
Training ()	The training process is similar to other models, such as convolutional neural networks or CNNs, recurrent neural networks or RNNs, transformers, and generative models. It uses the modified loss shape and other functions, such as validation and checkpoint saving, that help in the optimization.	`lr = 5 # learning rate` `optimizer = torch.optim.SGD(model.parameters(),lr=lr)` `scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 10000, gamma=0.9)` `def train(model: nn.Module,train_data) -> None:` `model.train() # turn on train mode` `total_loss = 0.` `log_interval = 10000` `start_time = time.time()` `num_batches = len(list(train_data)) // block_size` `for batch,srctgt in enumerate(train_data):` `src= srctgt[0]`

		`tgt= srctgt[1]` `logits = model(src,src_mask=None, key_padding_mask=None)` `logits_flat = logits.reshape(-1, logits.shape[-1])` `loss = loss_fn(logits_flat, tgt.reshape(-1))` `optimizer.zero_grad()` `loss.backward()` `torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)` `optimizer.step()` `total_loss += loss.item()` `if (batch % log_interval == 0 and batch > 0) or batch==42060:` `lr = scheduler.get_last_lr()[0]` `ms_per_batch = (time.time() - start_time) * 1000 / log_interval` `#cur_loss = total_loss / log_interval` `cur_loss = total_loss / batch` `ppl = math.exp(cur_loss)` `print(f'| epoch {epoch:3d} | {batch//block_size:5d}/{num_batches:5d} batches | '` `f'lr {lr:02.4f} | ms/batch {ms_per_batch:5.2f} | '` `f'loss {cur_loss:5.2f} | ppl {ppl:8.2f}')` `#total_loss = 0` `start_time = time.time()` `scheduler.step()`
Training: Validation function	The validation function is important to assess the model on a separate, invisible dataset during training to gauge generalization.	`def validate(model, validation_loader, loss_fn):` `   model.eval()` `   total_loss = 0` `   with torch.no_grad():` `     for src, tgt in validation_loader:` `       src, tgt = src.to(DEVICE), tgt.to(DEVICE)`

		logits = model(src) loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt.reshape(-1)) total_loss += loss.item() return total_loss / len(validation_loader)
Checkpoint saving function	The checkpoint saving function is useful for saving the model's state after certain intervals or under specific conditions, like improved validation performance.	def save_checkpoint(model, optimizer, filename="my_checkpoint.pth"): checkpoint = { "model_state_dict": model.state_dict(), "optimizer_state_dict": optimizer.state_dict(), } torch.save(checkpoint, filename)
Training: Evaluate function	The 'evaluate' function measures the performance of the model by computing its average loss in the validation dataset. However, the trained model is useful to generate inferences.	def evaluate(model: nn.Module, eval_data) -> float: model.eval() # turn on evaluation mode total_loss = 0. with torch.no_grad(): for src,tgt in eval_data: tgt = tgt.to(DEVICE) #seq_len = src.size(0) logits = model(src,src_mask=None, key_padding_mask=None) total_loss += loss_fn(logits.reshape(-1, logits.shape[-1]), tgt.reshape(-1)).item() return total_loss / (len(list(eval_data)) - 1)
Prompt()	Preparing an encoding prompt helps to create a process for text generation. This process serves as a starting point for the model to generate subsequent tokens. Once this prompt is tokenized, the decoder model can process and generate the next tokens based on the input.	def encode_prompt(prompt, block_size=BLOCK_SIZE): # Handle None prompt if prompt is None: prompt = '<pad>' * block_size else: tokens = tokenizer(prompt) number_of_tokens = len(tokens) # Adjust prompt length to fit block_size

		if number_of_tokens > block_size: tokens = tokens[-block_size:] # Keep last block_size tokens elif number_of_tokens < block_size: padding = ['<pad>'] * (block_size - number_of_tokens) tokens = padding + tokens # Prepend padding tokens prompt_indices = vocab(tokens) prompt_encoded = torch.tensor(prompt_indices, dtype=torch.int64).reshape(-1, 1) return prompt_encoded
Prompt encoding	Tokenized decoded prompt	prompt_encoded=encode_prompt("This is a prompt to get model generate next words.") prompt_encoded
Generate Function Step 1	The ‘generate’ function creates autoregressive text in the decoder model.	#Auto-regressive Language Model text generation def generate(model, prompt=None, max_new_tokens=500, block_size=BLOCK_SIZE, vocab=vocab, tokenizer=tokenizer): model.to(DEVICE) # Encode the input prompt prompt_encoded = encode_prompt(prompt).to(DEVICE) tokens = [] # Generate new tokens up to max_new_tokens for _ in range(max_new_tokens): # Decode the encoded prompt using the model's decoder logits = model(prompt_encoded) # Bring the sequence length to the first dimension logits = logits.transpose(0, 1) # Select the logits of the last token in the sequence

| | | ```
logit_prediction = logits[:, -1]

# Choose the most probable next token from the
logits(greedy decoding)
next_token_encoded = torch.argmax(logit_prediction,
dim=-1).reshape(-1, 1)

# If the next token is the end-of-sequence (EOS) token,
stop generation
if next_token_encoded.item() == EOS_IDX:
     break

# Append the next token to the prompt_encoded and keep
only the last 'block_size' tokens
prompt_encoded = torch.cat((prompt_encoded,
next_token_encoded), dim=0)[-block_size:]

# Convert the next token index to a token string using
the vocabulary
token_id = next_token_encoded.to('cpu').item()
tokens.append(vocab.get_itos()[token_id])

# Join the generated tokens into a single string and
return
return ' '.join(tokens)
``` |
| **Generate a token Step 2** | This function generates text using an auto-regressive language model. It iteratively predicts the next token in the sequence based on the previous tokens, using greedy decoding. The generation stops when either the maximum number of new tokens is reached or an end- | ```
#Auto-regressive Language Model text generation
def generate(model, prompt=None, max_new_tokens=500,
block_size=BLOCK_SIZE, vocab=vocab,
tokenizer=tokenizer):
model.to(DEVICE)

# Encode the input prompt
prompt_encoded = encode_prompt(prompt).to(DEVICE)
``` |

<table>
<tr>
<td></td>
<td>of-sequence token is predicted. Finally, it returns the generated text.</td>
<td><pre><code>tokens = []
Generate new tokens up to max_new_tokens
for _ in range(max_new_tokens):
Decode the encoded prompt using the model's decoder
logits = model.decoder(prompt_encoded,src_mask=None,
key_padding_mask=None)

Bring the sequence length to the first dimension
logits = logits.transpose(0, 1)

Select the logits of the last token in the sequence
logit_prediction = logits[:, -1]

Choose the most probable next token from the
logits(greedy decoding)
next_token_encoded = torch.argmax(logit_prediction,
dim=-1).reshape(-1, 1)

If the next token is the end-of-sequence (EOS) token,
stop generation
if next_token_encoded.item() == EOS_IDX:
     break

Append the next token to the prompt_encoded and keep
only the last 'block_size' tokens
prompt_encoded = torch.cat((prompt_encoded,
next_token_encoded), dim=0)[-block_size:]

Convert the next token index to a token string using
the vocabulary
token_id = next_token_encoded.to('cpu').item()
tokens.append(vocab.get_itos()[token_id])</code></pre></td>
</tr>
</table>

		`# Join the generated tokens into a single string and return` `return ' '.join(tokens)`
Generate Function Step 3:	This function generates text using an auto-regressive language model. It takes a pre-trained model, an optional prompt, and parameters for controlling text generation. It iteratively predicts the next token in the sequence based on the previous tokens. The generation stops when either the maximum number of new tokens is reached or an end-of-sequence token is predicted. Finally, it returns the generated text.	`#Auto-regressive Language Model text generation` `def generate(model, prompt=None, max_new_tokens=500, block_size=BLOCK_SIZE, vocab=vocab, tokenizer=tokenizer):` `model.to(DEVICE)` `# Encode the input prompt` `prompt_encoded = encode_prompt(prompt).to(DEVICE)` `tokens = []` `# Generate new tokens up to max_new_tokens` `for _ in range(max_new_tokens):` `# Decode the encoded prompt using the model's decoder` `logits = model.decoder(prompt_encoded,src_mask=None, key_padding_mask=None)` `# Bring the sequence length to the first dimension` `logits = logits.transpose(0, 1)` `# Select the logits of the last token in the sequence` `logit_prediction = logits[:, -1]` `# Choose the most probable next token from the logits(greedy decoding)` `next_token_encoded = torch.argmax(logit_prediction, dim=-1).reshape(-1, 1)` `# If the next token is the end-of-sequence (EOS) token, stop generation` `if next_token_encoded.item() == EOS_IDX:`

<table>
<tr><td></td><td></td><td>

```
    break

# Append the next token to the prompt_encoded and keep
only the last 'block_size' tokens
prompt_encoded = torch.cat((prompt_encoded,
next_token_encoded), dim=0)[-block_size:]

# Convert the next token index to a token string using
the vocabulary
token_id = next_token_encoded.to('cpu').item()
tokens.append(vocab.get_itos()[token_id])

# Join the generated tokens into a single string and
return
return ' '.join(tokens)
```

</td></tr>
<tr><td>Tokenization and vocabulary building</td><td>This code sets up the necessary infrastructure for tokenizing text data and converting it into numerical indices, facilitating subsequent NLP tasks like model training and evaluation.</td><td>

```
# Import the necessary libraries
tokenizer = get_tokenizer("basic_english")

# Define a function to yield tokenized samples
def yield_tokens(data_iter):
for label, data_sample in data_iter:
yield tokenizer(data_sample)

# Define special symbols and their indices
PAD_IDX, CLS_IDX, SEP_IDX, MASK_IDX, UNK_IDX = 0, 1, 2,
3, 4
special_symbols = ['[PAD]', '[CLS]', '[SEP]', '[MASK]',
'[UNK]']

# Split the data into training and testing sets using
the IMDB dataset
train_iter, test_iter = IMDB(split=('train', 'test'))
```

</td></tr>
</table>

		# Build the vocabulary from the training data vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials=special_symbols, special_first=True) # Set the default index of the vocabulary to UNK_IDX vocab.set_default_index(UNK_IDX) # Get the size of the vocabulary VOCAB_SIZE = len(vocab) text_to_index=lambda text: [vocab(token) for token in tokenizer(text)] index_to_en = lambda seq_en: " ".join([vocab.get_itos()[index] for index in seq_en])
Text Masking	This code defines a function Masking(token) which is responsible for applying masking to a token with a certain probability. If the mask decision is false (with an 80% chance), it returns the token with a '[PAD]' label. If masking is applied (with a 20% chance), it randomly selects between three cases.	def Masking(token): # Decide whether to mask this token (20% chance) mask = bernoulli_true_false(0.2) # If mask is False, immediately return with '[PAD]' label if not mask: return token, '[PAD]' # If mask is True, proceed with further operations # Randomly decide on an operation (50% chance each) random_opp = bernoulli_true_false(0.5) random_swich = bernoulli_true_false(0.5) # Case 1: If mask, random_opp, and random_swich are True if mask and random_opp and random_swich:

		`` # Replace the token with '[MASK]' and set label to a random token mask_label = index_to_en(torch.randint(0, VOCAB_SIZE, (1,))) token_ = '[MASK]' # Case 2: If mask and random_opp are True, but random_swich is False elif mask and random_opp and not random_swich: # Leave the token unchanged and set label to the same token token_ = token mask_label = token # Case 3: If mask is True, but random_opp is False else: # Replace the token with '[MASK]' and set label to the original token token_ = '[MASK]' mask_label = token return token_, mask_label ```
MLM preparations	This code defines a function Masking(token) which is responsible for applying masking to a token. It decides whether to mask the token with a 20% chance. If not, it returns the token with a '[PAD]' label. If masking is applied, it randomly chooses between three cases: Case 1: It replaces the token with '[MASK]' and assigns a random token as the label.	``` def Masking(token): # Decide whether to mask this token (20% chance) mask = bernoulli_true_false(0.2) # If mask is False, immediately return with '[PAD]' label if not mask: return token, '[PAD]' # If mask is True, proceed with further operations # Randomly decide on an operation (50% chance each) ```

	Case 2: It retains the token unchanged and assigns the same token as the label. Case 3: It replaces the token with '[MASK]' and assigns the original token as the label. The choice between these cases is determined by two independent 50% chances (random_opp and random_swich). Finally, it returns the modified token and its corresponding label.	`random_opp = bernoulli_true_false(0.5)` `random_swich = bernoulli_true_false(0.5)` `# Case 1: If mask, random_opp, and random_swich are True` `if mask and random_opp and random_swich:` `# Replace the token with '[MASK]' and set label to a random token` `mask_label = index_to_en(torch.randint(0, VOCAB_SIZE, (1,)))` `token_ = '[MASK]'` `# Case 2: If mask and random_opp are True, but random_swich is False` `elif mask and random_opp and not random_swich:` `# Leave the token unchanged and set label to the same token` `token_ = token` `mask_label = token` `# Case 3: If mask is True, but random_opp is False` `else:` `# Replace the token with '[MASK]' and set label to the original token` `token_ = '[MASK]'` `mask_label = token` `return token_, mask_label`
NSP preparations	This code defines a function process_for_nsp which prepares inputs for training BERT for the next sentence prediction (NSP) task. It takes two inputs: input_sentences, a list of sentences, and input_masked_labels, a list of labels	`def process_for_nsp(input_sentences, input_masked_labels):` `# Verify that both input lists are of the same length and have a sufficient number of sentences`

	corresponding to masked tokens in the sentences.	`if len(input_sentences) < 2:` `     raise ValueError("Must have two same number of items.")` `if len(input_sentences) != len(input_masked_labels):` `     raise ValueError("Both lists must have the same number of items.")` `bert_input = []` `bert_label = []` `is_next = []` `available_indices = list(range(len(input_sentences)))` `while len(available_indices) >= 2:` `if random.random() < 0.5:` `# Choose two consecutive sentences to simulate the 'next sentence' scenario` `index = random.choice(available_indices[:-1]) # Exclude the last index` `# append list and add '[CLS]' and '[SEP]' tokens` `bert_input.append([['[CLS]']+input_sentences[index]+ ['[SEP]'],input_sentences[index + 1]+ ['[SEP]']])` `bert_label.append([['[PAD]']+input_masked_labels[index]+ ['[PAD]'], input_masked_labels[index + 1]+ ['[PAD]']])` `is_next.append(1) # Label 1 indicates these sentences are consecutive` `# Remove the used indices` `available_indices.remove(index)` `if index + 1 in available_indices:` `     available_indices.remove(index + 1)` `else:` `# Choose two random distinct sentences to simulate the 'not next sentence' scenario`

		indices = random.sample(available_indices, 2) bert_input.append([['[CLS]']+input_sentences[indices[0]]+['[SEP]'],input_sentences[indices[1]]+ ['[SEP]']]) bert_label.append([['[PAD]']+input_masked_labels[indices[0]]+['[PAD]'], input_masked_labels[indices[1]]+['[PAD]']]) is_next.append(0) # Label 0 indicates these sentences are not consecutive # Remove the used indices available_indices.remove(indices[0]) available_indices.remove(indices[1]) return bert_input, bert_label, is_next
Creating training-ready inputs for BERT	This code defines a function prepare_bert_final_inputs which prepares inputs for training a BERT model. It takes bert_inputs, bert_labels, and is_nexts as inputs. The function pads the inputs and labels with [PAD] tokens to ensure they are of equal length, creates segment labels for each pair of sentences, and converts the inputs, labels, and segment labels into tensors if to_tensor is set to True. Finally, it returns the processed inputs, labels, segment labels, and is_nexts.	def prepare_bert_final_inputs(bert_inputs, bert_labels, is_nexts, to_tensor=True): def zero_pad_list_pair(pair_, pad='[PAD]'): pair = deepcopy(pair_) max_len = max(len(pair[0]), len(pair[1])) # Append [PAD] to each sentence in the pair until the maximum length is reached pair[0].extend([pad] * (max_len - len(pair[0]))) pair[1].extend([pad] * (max_len - len(pair[1]))) return pair[0], pair[1] # Flatten the tensor flatten = lambda l: [item for sublist in l for item in sublist] # Transform tokens to vocab indices tokens_to_index = lambda tokens: [vocab[token] for token in tokens]

		bert_inputs_final, bert_labels_final, segment_labels_final, is_nexts_final = [], [], [], [] for bert_input, bert_label, is_next in zip(bert_inputs, bert_labels, is_nexts): # Create segment labels for each pair of sentences segment_label = [[1] * len(bert_input[0]), [2] * len(bert_input[1])] # Zero-pad the bert_input, bert_label, and segment_label bert_input_padded = zero_pad_list_pair(bert_input) bert_label_padded = zero_pad_list_pair(bert_label) segment_label_padded = zero_pad_list_pair(segment_label, pad=0) # Convert to tensors if to_tensor: # Flatten the padded inputs and labels, transform tokens to their corresponding vocab indices, and convert them to tensors bert_inputs_final.append(torch.tensor(tokens_to_index(flatten(bert_input_padded)), dtype=torch.int64)) bert_labels_final.append(torch.tensor(tokens_to_index(flatten(bert_label_padded)), dtype=torch.int64)) segment_labels_final.append(torch.tensor(flatten(segment_label_padded), dtype=torch.int64)) is_nexts_final.append(is_next) else: # Flatten the padded inputs and labels bert_inputs_final.append(flatten(bert_input_padded)) bert_labels_final.append(flatten(bert_label_padded)) segment_labels_final.append(flatten(segment_label_padded))

<table>
<tr>
<td></td>
<td></td>
<td>

```
is_nexts_final.append(is_next)

return bert_inputs_final, bert_labels_final,
segment_labels_final, is_nexts_final
```

</td>
</tr>
<tr>
<td>Creating training-ready CSV file from IMDB</td>
<td>This code writes the processed Internet Movie Database or IMDB data to a CSV file.</td>
<td>

```
csv_file_path = 'train_bert_data_new.csv'

# Open the CSV file for writing
with open(csv_file_path, mode='w', newline='',
encoding='utf-8') as file:
csv_writer = csv.writer(file)

# Write the header row
csv_writer.writerow(['Original Text', 'BERT Input',
'BERT Label', 'Segment Label', 'Is Next'])

# Wrap train_iter with tqdm for a progress bar
for n, (_, sample) in enumerate(tqdm(train_iter,
desc="Processing samples")):
# Tokenize the sample input
tokens = tokenizer(sample)

# Create MLM inputs and labels
bert_input, bert_label = prepare_for_mlm(tokens,
include_raw_tokens=False)

# Skip samples with insufficient input length
if len(bert_input) < 2:
     continue

# Create NSP pairs, token labels, and is_next label
bert_inputs, bert_labels, is_nexts =
process_for_nsp(bert_input, bert_label)
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td><pre><code># Add zero-paddings, map tokens to vocab indices, and
create segment labels
bert_inputs, bert_labels, segment_labels, is_nexts =
prepare_bert_final_inputs(bert_inputs, bert_labels,
is_nexts)

Convert tensors to lists and then convert lists to
JSON-formatted strings
for bert_input, bert_label, segment_label, is_next in
zip(bert_inputs, bert_labels, segment_labels, is_nexts):
bert_input_str = json.dumps(bert_input.tolist())
bert_label_str = json.dumps(bert_label.tolist())
segment_label_str = ','.join(map(str,
segment_label.tolist()))

Write the data to a CSV file row-by-row
csv_writer.writerow([sample, bert_input_str,
bert_label_str, segment_label_str, is_next])</code></pre></td>
</tr>
<tr>
<td>__init__</td>
<td>Used to initialize objects of a class. It is also called a constructor.</td>
<td><pre><code>from pyspark.sql import SparkSession
spark =
SparkSession.builder.appName("MyApp").getOrCreate()</code></pre></td>
</tr>
<tr>
<td>__len__</td>
<td>essentially used to implement the built-in len() function. Whenever you call len(), Python internally invokes the __len__ magic method.</td>
<td>def __len__(self):

return len(self.data)</td>
</tr>
<tr>
<td>__getitem__</td>
<td>Used to define the behavior of retrieving items from an object.</td>
<td><pre><code>def __getitem__(self, idx):
        return self.data[idx]</code></pre></td>
</tr>
<tr>
<td>torch.tensor(...)</td>
<td>Creates a PyTorch tensor from the Python object obtained from the JSON string. It converts the Python object into a PyTorch tensor.</td>
<td><pre><code>torch.tensor(json.loads(row['BERT Input']))</code></pre></td>
</tr>
</table>

Is_next	A PyTorch tensor created from a value stored in a DataFrame row. Specifically, it's created from the value associated with the key 'Is Next'	`is_next = torch.tensor(row['Is Next'], dtype=torch.long)`
collate_batch	Responsible for collating individual samples into batches.	`def collate_batch(batch):` `label_list, text_list, lengths = [], [], []` `for _label, _text in batch:` `label_list.append(label_pipeline(_label))` `processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)` `text_list.append(processed_text)` `lengths.append(processed_text.size(0))` `if CONFIG_USE_ROCM:` `    label_list = torch.tensor(label_list, device='cuda')` `    lengths = torch.tensor(lengths, device='cuda')` `else:` `label_list = torch.tensor(label_list)` `    lengths = torch.tensor(lengths)` `padded_text_list = nn.utils.rnn.pad_sequence(text_list, batch_first=True)` `padded_text_list.to('cuda')` `#code.interact(local=locals())` `return padded_text_list, label_list, lengths`
forward	Defines the forward pass computation, which includes applying the various embedding layers and dropout during training.	`def forward(self, bert_inputs, segment_labels=False):` `my_embeddings = self.token_embedding(bert_inputs)` `if self.train:` `x = self.dropout(my_embeddings`

		<pre>+ self.positional_encoding(my_embeddings) + self.segment_embedding(segment_labels)) else: x = my_embeddings + self.positional_encoding(my_embeddings) return x</pre>
torch.no_grad()	Context manager provided by PyTorch that turns off gradients during validation or evaluation to save memory and computations.	with torch.no_grad(): # Turn off gradients for validation, # saves memory and computations for batch in dataloader: bert_inputs, bert_labels, segment_labels, is_nexts = [b.to(device) for b in batch]
evaluate	Used for evaluating the BERT model's performance on the test dataset. It calculates the average loss over all batches in the test dataset and prints the average loss, average next sentence loss, and average mask loss.	<pre>def evaluate(dataloader=test_dataloader, model=model, loss_fn=loss_fn, device=device): model.eval() # Turn off dropout and other training- specific behaviors total_loss = 0 total_next_sentence_loss = 0 total_mask_loss = 0 total_batches = 0</pre>

<table>
<tr><td>Adam</td><td>Initializes the Adam optimizer, which is a variant of stochastic gradient descent (SGD). It's commonly used for optimizing neural network models.</td><td><pre><code>optimizer = Adam(model.parameters(), lr=1e-4,
weight_decay=0.01,

                                              betas=(0.9,
0.999))</code></pre></td></tr>
<tr><td>zero_grad()</td><td>Used to zero out the gradients of all parameters of the model. It's typically called before performing the backward pass to avoid accumulating gradients from previous iterations.</td><td><pre><code>import torch
from torch.autograd import Variable
import torch.optim as optim
def linear_model(x, W, b):
    return torch.matmul(x, W) + b
data, targets = ...
a = Variable(torch.randn(4, 3), requires_grad=True)
b = Variable(torch.randn(3), requires_grad=True)
optimizer = optim.Adam([a, b])
for sample, target in zip(data, targets):
    optimizer.zero_grad()
    output = linear_model(sample, W, b)
    loss = (output - target) ** 2
    loss.backward()
    optimizer.step()</code></pre></td></tr>
<tr><td>backward()</td><td>Computes gradients of the loss with respect to the model parameters.</td><td><pre><code>import torch
from torch.autograd import Variable
import torch.optim as optim
def linear_model(x, W, b):
    return torch.matmul(x, W) + b
data, targets = ...
a = Variable(torch.randn(4, 3), requires_grad=True)
b = Variable(torch.randn(3), requires_grad=True)
optimizer = optim.Adam([a, b])
for sample, target in zip(data, targets):
    optimizer.zero_grad()
    output = linear_model(sample, W, b)
    loss = (output - target) ** 2</code></pre></td></tr>
</table>

		`loss.backward()` `optimizer.step()`
torch.nn.utils.clip_grad_norm_	Used for gradient clipping, which is a technique to prevent the exploding gradient problem during training.	`torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)`
step()	Updates the parameters of the model using the gradients computed during backpropagation.	`optimizer.step()`
torch.save	Used to save the model's state dictionary to a file.	`torch.save(model.state_dict(), model_save_path)`
plt.plot	Used to plot data points on a graph. It takes the x-values, y-values, and optional arguments to customize the plot, such as line style, color, and label.	`plt.plot(range(1, num_epochs + 1), train_losses, label='Training Loss')`
plt.xlabel	Used to set the label for the x-axis of the plot.	`plt.xlabel('Epoch')`
plt.ylabel	Used to set the label for the y-axis of the plot.	`plt.ylabel('Loss')`
plt.title	Used to set the title of the plot. It specifies the text that will be displayed as the title above the plot.	`plt.title('Training and Evaluation Loss')`
plt.legend	Used to add a legend to the plot. It displays labels associated with each plot line,	`plt.legend()`
plt.show	Used to display the plot on the screen or in the output of the script.	`plt.show()`
predict_nsp	A function that takes two sentences, a BERT model, and a tokenizer as input. It tokenizes the input sentences using the tokenizer, then	`sentence1 = "The cat is sitting on the chair."` `sentence2 = "It is a rainy day"`

	feeds the tokenized inputs to the BERT model to predict whether the second sentence follows the first one (Next Sentence Prediction task). The function returns a string indicating whether the second sentence follows the first one or not based on the model's prediction.	`print(predict_nsp(sentence1, sentence2, model, tokenizer))`
predict_mlm	Takes an input sentence, a BERT model, and a tokenizer as input. It tokenizes the input sentence using the tokenizer and converts it into token IDs. Then, it creates dummy segment labels filled with zeros and feeds the input tokens and segment labels to the BERT model. The function extracts the position of the [MASK] token and retrieves the predicted index for the [MASK] token from the model's predictions. Finally, it replaces the [MASK] token in the original sentence with the predicted token and returns the predicted sentence.	`def predict_mlm(sentence, model, tokenizer):` `# Tokenize the input sentence and convert to token IDs,` `# including special tokens` `inputs = tokenizer(sentence, return_tensors="pt")` `tokens_tensor = inputs.input_ids`
generate_square_subsequent_mask	Generates a square subsequent mask for self-attention mechanisms in transformer-based models.	`def generate_square_subsequent_mask(sz,device=DEVICE):` `mask = (torch.triu(torch.ones((sz, sz), device=device)) == 1).transpose(0, 1)` `mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))` `return mask`

create_mask	Creates masks for the source and target sequences, as well as padding masks for both sequences.	`def create_mask(src, tgt,device=DEVICE):` `src_seq_len = src.shape[0]` `tgt_seq_len = tgt.shape[0]` `tgt_mask = generate_square_subsequent_mask(tgt_seq_len)` `src_mask = torch.zeros((src_seq_len, src_seq_len),device=DEVICE).type(torch.bool)` `src_padding_mask = (src == PAD_IDX).transpose(0, 1)` `tgt_padding_mask = (tgt == PAD_IDX).transpose(0, 1)` `return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask`
encode	Responsible for encoding the input source sequence into a fixed-dimensional representation that captures the contextual information of the input sequence.	`def encode(self, src: Tensor, src_mask: Tensor):` `src_embedded = self.src_tok_emb(src)` `src_pos_encoded = self.positional_encoding(src_embedded)` `return self.transformer.encoder(src_pos_encoded, src_mask)`
decode	Generates the output sequence based on the encoded source sequence and the target sequence.	`def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):` `tgt_embedded = self.tgt_tok_emb(tgt)` `tgt_pos_encoded = self.positional_encoding(tgt_embedded)`

		`return self.transformer.decoder(tgt_pos_encoded, memory, tgt_mask)`
train_epoch	Represents a training epoch in the training loop. It takes the model, optimizer, and training dataloader as input arguments and returns the average loss over the epoch.	`def train_epoch(model, optimizer,train_dataloader):` `model.train()` `losses = 0` `for src, tgt in train_dataloader:` `src = src.to(DEVICE)` `tgt = tgt.to(DEVICE)` `tgt_input = tgt[:-1, :]` `src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_input)` `src_mask = src_mask.to(DEVICE)` `tgt_mask= tgt_mask.to(DEVICE)` `src_padding_mask= src_padding_mask.to(DEVICE)` `tgt_padding_mask =tgt_padding_mask.to(DEVICE)` `logits = model(src, tgt_input, src_mask, tgt_mask,src_padding_mask, tgt_padding_mask, src_padding_mask)` `logits = logits.to(DEVICE)` `optimizer.zero_grad()` `tgt_out = tgt[1:, :]` `loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt_out.` `Reshape(-1))` `loss.backward()` `optimizer.step()` `losses += loss.item()` `return losses / len(list(train_dataloader))`

greedy_decode	Performs greedy decoding to generate an output sequence using the trained transformer model.	`def greedy_decode(model, src, src_mask, max_len, start_symbol):` `src = src.to(DEVICE)` `src_mask = src_mask.to(DEVICE)` `memory = model.encode(src, src_mask)` `ys = torch.ones(1, 1).fill_(start_symbol).type(torch.long).to(DEVICE)` `for i in range(max_len-1):` `     memory = memory.to(DEVICE)` `tgt_mask = (generate_square_subsequent_mask(ys.size(0)).type(torch.bool)).to(DEVICE)` `out = model.decode(ys, memory, tgt_mask)` `out = out.transpose(0, 1)` `prob = model.generator(out[:, -1])` `_, next_word = torch.max(prob, dim=1)` `next_word = next_word.item()` `ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=0)` `if next_word == EOS_IDX:` `     break` `return ys`
translate(model: torch.nn.Module, src_sentence: str)	Translates a given source sentence into the target language using the provided PyTorch model.	`def translate(model: torch.nn.Module, src_sentence: str):` `    model.eval()` `    src = text_transform[SRC_LANGUAGE](src_sentence).view(-1, 1)` `    num_tokens = src.shape[0]` `    src_mask = (torch.zeros(num_tokens, num_tokens)).type(torch.bool)`

		tgt_tokens = greedy_decode(model, src, src_mask, max_len=num_tokens + 5, start_symbol=BOS_IDX).flatten() return " ".join(vocab_transform[TGT_LANGUAGE].lookup_tokens(list(tgt_tokens.cpu().numpy()))).replace("<bos>", "").replace("<eos>", "")