**TICS –II**

**ASSIGNMENT**

**HUSSAIN ALI**

**FA21-BCS-066**

**BCS-8A**

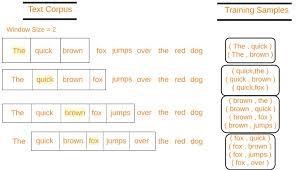
**-------------------------------------------------**

**2.2**

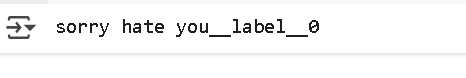
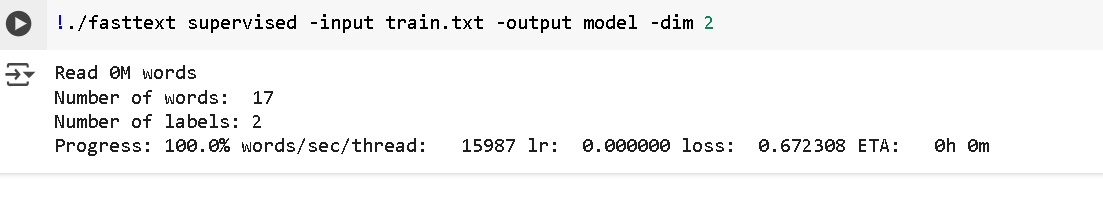
## Section 2.2: :

## Word2Vec

**Model Architecture Diagram:**



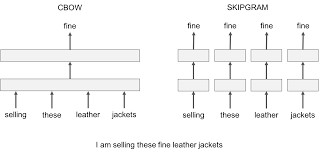
****

****

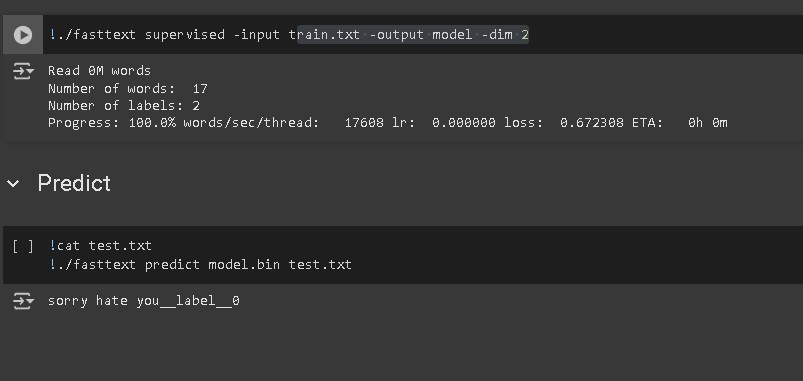
## Section 2.3:

## FastText – Subword Representations

**Model Architecture Diagram:**



**Execution results**



**Understanding the Code**

1. **Preparing Training and Testing Data** f = open('train.txt', 'w')

f.write('\_\_label\_\_1 i love you\n')

f.write('\_\_label\_\_1 he loves me\n')

f.write('\_\_label\_\_1 she likes baseball\n')

f.write('\_\_label\_\_0 i hate you\n')

f.write('\_\_label\_\_0 sorry for that\n')

f.write('\_\_label\_\_0 this is awful')

f.close()

* + A file named **train.txt** is created.
  + Each line in this file is labeled using **FastText's format**, where:
    - \_\_label\_\_1 represents **positive sentiment** (e.g., love, likes).
    - \_\_label\_\_0 represents **negative sentiment** (e.g., hate, awful, sorry).
  + This labeled data is used to train a supervised classification model.

1. **Creating Test Data**

f = open('test.txt', 'w')

f.write('sorry hate you')

f.close()

• A file named **test.txt** is created, containing a sentence for classification.

1. **Training the FastText Model**

!./fasttext supervised -input train.txt -output model -dim 2 • The supervised mode is used to train a classification model.

* + -input train.txt specifies the training data.
  + -output model saves the trained model as model.bin and model.vec.
  + -dim 2 sets the feature vector dimension to 2 (FastText embeddings are used for text representation).

1. **Checking the Test Data**

!cat test.txt

* + This command simply prints the contents of test.txt, which is: • sorry hate you

1. **Making Predictions with FastText**

!./fasttext predict model.bin test.txt

* + The trained model (model.bin) is used to **predict** the label for test.txt.
  + The output will be either \_\_label\_\_1 (positive) or \_\_label\_\_0 (negative), based on the trained data.

**Running This Code**

If you want to run this code yourself, follow these steps: **Step 1: Install FastText** pip install fasttext

or compile it manually from [FastText GitHub.](https://github.com/facebookresearch/fastText)

**Step 2: Create the Python Script**

Save the following as train\_test\_fasttext.py:

import fasttext

# Create training file with open('train.txt', 'w') as f:

f.write('\_\_label\_\_1 i love you\n')

f.write('\_\_label\_\_1 he loves me\n')

f.write('\_\_label\_\_1 she likes baseball\n')

f.write('\_\_label\_\_0 i hate you\n')

f.write('\_\_label\_\_0 sorry for that\n')

f.write('\_\_label\_\_0 this is awful\n')

# Create test file with open('test.txt', 'w') as f:

f.write('sorry hate you')

# Train the model model = fasttext.train\_supervised(input="train.txt", dim=2)

# Save the model model.save\_model("model.bin")

# Read and predict with open("test.txt", "r") as f: sentence = f.readline().strip() print("Test sentence:", sentence)

# Predict the label prediction = model.predict(sentence) print("Predicted label:", prediction[0][0]) **Step 3: Run the Script**

python train\_test\_fasttext.py

**Expected Output**

Test sentence: sorry hate you

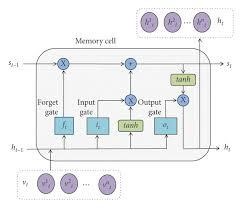
Predicted label: \_\_label\_\_0

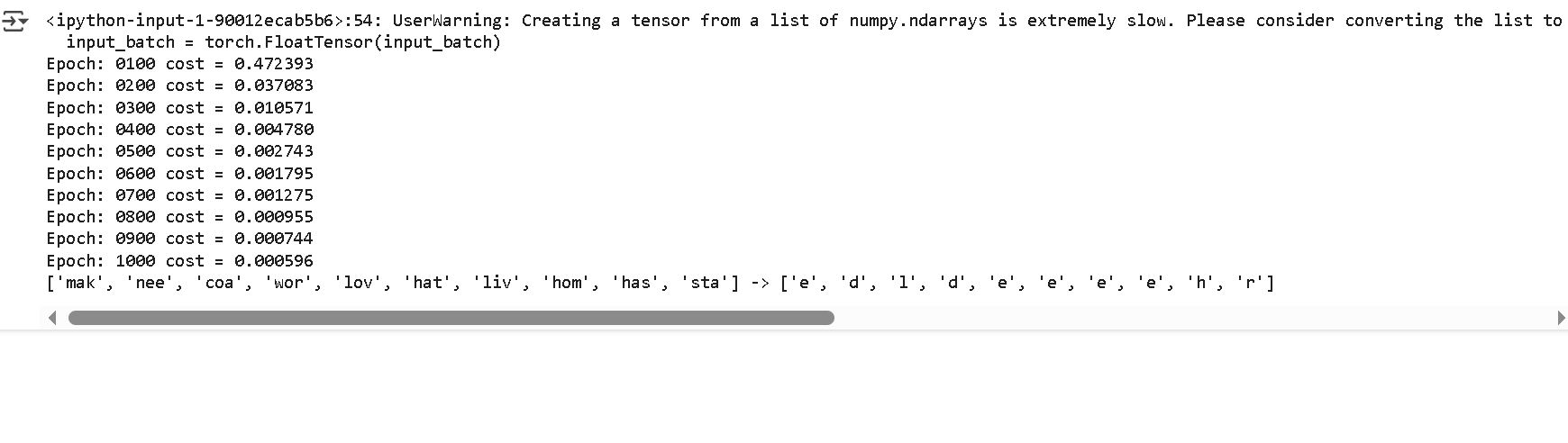
Since the words "sorry" and "hate" were used in negative examples, FastText predicts \_\_label\_\_0 (negative sentiment).

## Section 4.2:

## LSTM – The Gated Memory Cell

**Model architecture diagram:**



****

**Understanding the code:**

**Step-by-Step Explanation 1. Import Required Libraries** import numpy as np import torch import torch.nn as nn

import torch.optim as optim

* numpy is used for numerical operations.
* torch is the main PyTorch library.
* torch.nn contains neural network layers.
* torch.optim provides optimization algorithms (Adam in this case).

1. **Define the make\_batch Function** def make\_batch():

input\_batch, target\_batch = [], []

for seq in seq\_data:

input = [word\_dict[n] for n in seq[:-1]] # 'm', 'a' , 'k' is input target = word\_dict[seq[-1]] # 'e' is target input\_batch.append(np.eye(n\_class)[input]) target\_batch.append(target)

return input\_batch, target\_batch

* + This function converts the words into **one-hot encoded inputs** and their **corresponding targets**.
  + Example:

o "make" → Input: ['m', 'a', 'k'], Target: 'e' o "coal" → Input: ['c', 'o', 'a'], Target: 'l'

* + The inputs are one-hot encoded using np.eye(n\_class)[input].

1. **Define the LSTM Model (TextLSTM)** class TextLSTM(nn.Module):

def \_\_init\_\_(self):

super(TextLSTM, self).\_\_init\_\_()

self.lstm = nn.LSTM(input\_size=n\_class, hidden\_size=n\_hidden) self.W = nn.Linear(n\_hidden, n\_class, bias=False) self.b = nn.Parameter(torch.ones([n\_class]))

def forward(self, X):

input = X.transpose(0, 1) # X: [n\_step, batch\_size, n\_class]

hidden\_state = torch.zeros(1, len(X), n\_hidden) # [1, batch\_size, n\_hidden] cell\_state = torch.zeros(1, len(X), n\_hidden) # [1, batch\_size, n\_hidden]

outputs, (\_, \_) = self.lstm(input, (hidden\_state, cell\_state)) outputs = outputs[-1] # Take the last output (final time step) model = self.W(outputs) + self.b # Final prediction return model

* **nn.LSTM(input\_size, hidden\_size)**:
  + input\_size = n\_class (number of characters in vocabulary).
  + hidden\_size = n\_hidden (number of hidden units in LSTM).
* **Forward Pass**:
  + input is transposed to match LSTM’s expected shape [n\_step, batch\_size, n\_class]. o Initializes hidden\_state and cell\_state as zeros. o Feeds the sequence into the LSTM and extracts the last hidden state.
  + Passes the output through a **linear layer** (self.W) for classification.

**4. Model Training** if \_\_name\_\_ == '\_\_main\_\_':

n\_step = 3 # Number of time steps (sequence length) n\_hidden = 128 # Number of hidden units

char\_arr = [c for c in 'abcdefghijklmnopqrstuvwxyz'] word\_dict = {n: i for i, n in enumerate(char\_arr)} number\_dict = {i: w for i, w in enumerate(char\_arr)} n\_class = len(word\_dict) # Number of unique characters

seq\_data = ['make', 'need', 'coal', 'word', 'love', 'hate', 'live', 'home', 'hash', 'star']

model = TextLSTM()

criterion = nn.CrossEntropyLoss() optimizer = optim.Adam(model.parameters(), lr=0.001)

input\_batch, target\_batch = make\_batch() input\_batch = torch.FloatTensor(input\_batch) target\_batch = torch.LongTensor(target\_batch)

for epoch in range(1000): optimizer.zero\_grad()

output = model(input\_batch) loss = criterion(output, target\_batch) if (epoch + 1) % 100 == 0:

print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.6f}'.format(loss))

loss.backward()

optimizer.step()

* Converts input\_batch to **float tensor** and target\_batch to **long tensor**.
* Runs training for **1000 epochs**, updating weights using **Adam optimizer**.
* Prints the loss every 100 epochs.

**5. Testing the Model** inputs = [sen[:3] for sen in seq\_data]

predict = model(input\_batch).data.max(1, keepdim=True)[1]

print(inputs, '->', [number\_dict[n.item()] for n in predict.squeeze()])

* Extracts the **first three letters** of each word.
* Passes them through the trained model.
* Finds the predicted character (index with the highest probability).
* Converts predictions back to characters using number\_dict.

**Expected Output**

After training, the model should predict the **fourth letter** correctly:

Epoch: 0100 cost = 2.349817

Epoch: 0200 cost = 1.925671 Epoch: 0300 cost = 1.478965

...

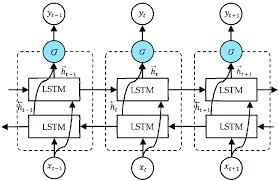
Epoch: 1000 cost = 0.003245

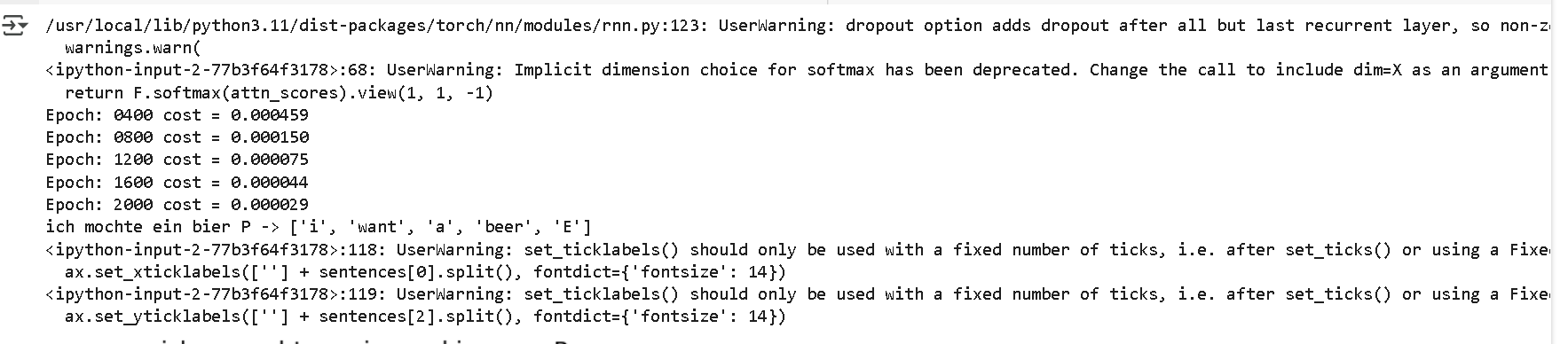
['mak', 'nee', 'coa', 'wor', 'lov', 'hat', 'liv', 'hom', 'has', 'sta']

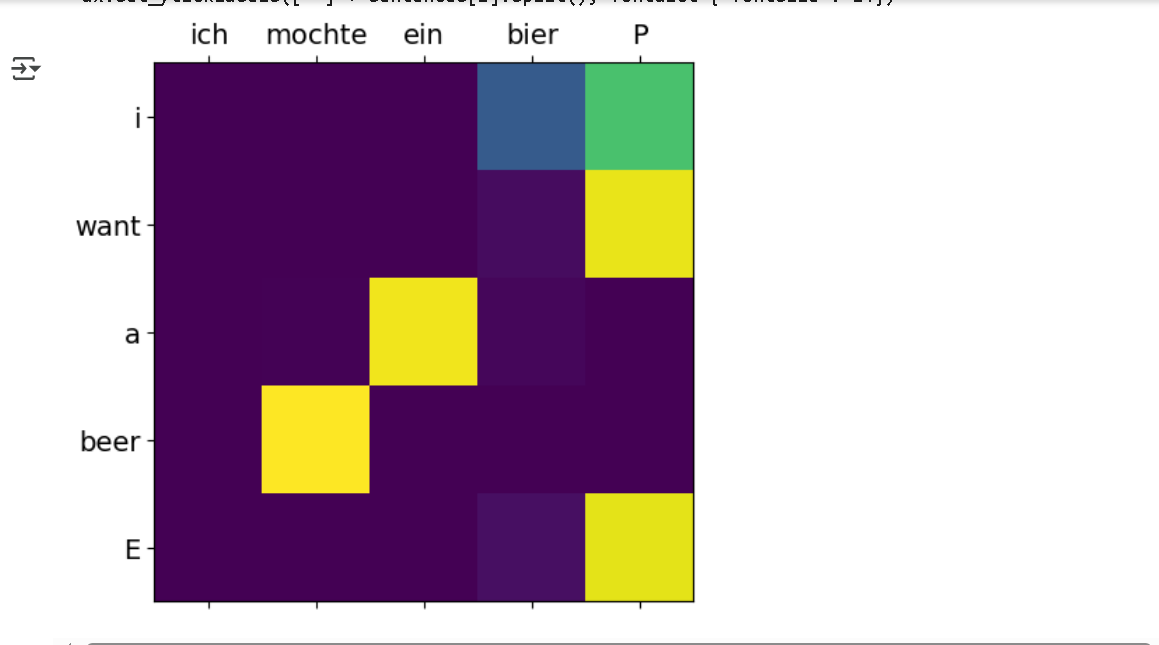
-> ['e', 'd', 'l', 'd', 'e', 'e', 'e', 'e', 'h', 'r']

**Section 4.3:**

**Model Architecture diagram:**



****

****

**Section 5.3:**

**Code execution results:**



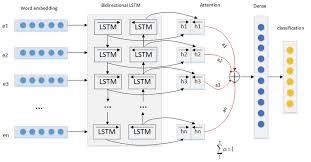
**Understanding the code:**

1. **Dataset and Preprocessing** 
   * The dataset consists of a simple translation example:
     + **Input sentence (German):** 'ich mochte ein bier P' o **Decoder input (English with start token):** 'S i want a beer'
     + **Target sentence (English with end token):** 'i want a beer E'
   * The words are converted into one-hot vectors using a dictionary (word\_dict).
2. **Model Architecture** 
   * **Encoder (RNN):** Encodes the input sentence into a hidden state.
   * **Decoder (RNN with Attention):** Generates the translated output step by step.
   * **Attention Mechanism:** Computes attention scores for each encoder output relative to the current decoder state.
3. **Training Process** 
   * The model is trained using **CrossEntropyLoss** and **Adam optimizer**.
   * Training runs for **2000 epochs**, printing the loss every **400 epochs**.
4. **Testing and Visualization** 
   * The trained model is tested on a test input ('SPPPP' for padding).
   * The output predictions are mapped back to words.
   * The attention matrix is visualized to show which input words influence the output at each step.

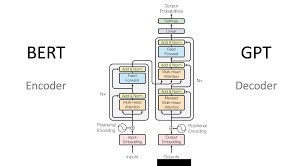
**Potential Enhancements**

* + **Use a larger dataset** for better generalization.
  + **Replace RNN with LSTM or Transformer** for improved performance.
  + **Train on GPU** for faster computation.

**Bi-LSTM with Attention for Sentiment Classification**



**Section 6.1: The Transformer Architecture**



**Understanding the code:**

1. **Overview of Key Components**

The implementation includes:

* + - **Positional Encoding:** Adds position information to input embeddings.
    - **Multi-Head Attention:** Allows the model to focus on different parts of the input sequence.
    - **Feed-Forward Network:** Processes the transformed embeddings.
    - **Encoder-Decoder Architecture:** Standard structure used for sequence-to-sequence tasks.

1. **Breaking Down the Code** 
   1. **Data Preparation**

The make\_batch() function creates tokenized input and output batches. Symbols:

* + - S: Start of decoding input.
    - E: Start of decoding output.
    - P: Padding for sequences shorter than the max length. def make\_batch():

input\_batch = [[src\_vocab[n] for n in sentences[0].split()]] output\_batch = [[tgt\_vocab[n] for n in sentences[1].split()]] target\_batch = [[tgt\_vocab[n] for n in sentences[2].split()]]

return torch.LongTensor(input\_batch), torch.LongTensor(output\_batch), torch.LongTensor(target\_batch)

* + - Converts words into indices using src\_vocab and tgt\_vocab.
    - Returns three tensors: **input**, **output**, and **target**.

* 1. **Positional Encoding**

Transformer does not use recurrence; instead, it uses **sinusoidal positional encoding** to retain order information. def get\_sinusoid\_encoding\_table(n\_position, d\_model):

def cal\_angle(position, hid\_idx):

return position / np.power(10000, 2 \* (hid\_idx // 2) / d\_model) def get\_posi\_angle\_vec(position):

return [cal\_angle(position, hid\_j) for hid\_j in range(d\_model)]

sinusoid\_table = np.array([get\_posi\_angle\_vec(pos\_i) for pos\_i in range(n\_position)]) sinusoid\_table[:, 0::2] = np.sin(sinusoid\_table[:, 0::2]) sinusoid\_table[:, 1::2] = np.cos(sinusoid\_table[:, 1::2])

return torch.FloatTensor(sinusoid\_table)

* sin() applies to even indices.
* cos() applies to odd indices.
* Helps Transformer understand **relative positioning** of tokens.

**2.3. Attention Mechanisms**

The Transformer heavily relies on **attention mechanisms**.

1. **Scaled Dot-Product Attention** class ScaledDotProductAttention(nn.Module):

def \_\_init\_\_(self):

super(ScaledDotProductAttention, self).\_\_init\_\_()

def forward(self, Q, K, V, attn\_mask): scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d\_k) scores.masked\_fill\_(attn\_mask, -1e9) attn = nn.Softmax(dim=-1)(scores) context = torch.matmul(attn, V) return context, attn

* + Computes attention scores using the formula:

[ \text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d\_k}}\right) V ]

* + attn\_mask prevents attention to **padding tokens**.
  + Softmax normalizes scores to a probability distribution.

1. **Multi-Head Attention** class MultiHeadAttention(nn.Module):

def \_\_init\_\_(self):

super(MultiHeadAttention, self).\_\_init\_\_() self.W\_Q = nn.Linear(d\_model, d\_k \* n\_heads) self.W\_K = nn.Linear(d\_model, d\_k \* n\_heads) self.W\_V = nn.Linear(d\_model, d\_v \* n\_heads) self.linear = nn.Linear(n\_heads \* d\_v, d\_model) self.layer\_norm = nn.LayerNorm(d\_model)

def forward(self, Q, K, V, attn\_mask): residual, batch\_size = Q, Q.size(0) q\_s = self.W\_Q(Q).view(batch\_size, -1, n\_heads, d\_k).transpose(1,2) k\_s = self.W\_K(K).view(batch\_size, -1, n\_heads, d\_k).transpose(1,2) v\_s = self.W\_V(V).view(batch\_size, -1, n\_heads, d\_v).transpose(1,2)

attn\_mask = attn\_mask.unsqueeze(1).repeat(1, n\_heads, 1, 1) context, attn = ScaledDotProductAttention()(q\_s, k\_s, v\_s, attn\_mask)

context = context.transpose(1, 2).contiguous().view(batch\_size, -1, n\_heads \* d\_v) output = self.linear(context)

return self.layer\_norm(output + residual), attn

* Uses multiple attention heads to focus on **different parts** of the sequence.
* Final output is normalized and **residual connections** are used.

**2.4. Feed-Forward Network** class PoswiseFeedForwardNet(nn.Module):

def \_\_init\_\_(self):

super(PoswiseFeedForwardNet, self).\_\_init\_\_() self.conv1 = nn.Conv1d(in\_channels=d\_model, out\_channels=d\_ff, kernel\_size=1) self.conv2 = nn.Conv1d(in\_channels=d\_ff, out\_channels=d\_model, kernel\_size=1) self.layer\_norm = nn.LayerNorm(d\_model)

def forward(self, inputs):

residual = inputs output = nn.ReLU()(self.conv1(inputs.transpose(1, 2))) output = self.conv2(output).transpose(1, 2) return self.layer\_norm(output + residual)

* Two linear layers with **ReLU activation**.
* Adds residual connections for **better gradient flow**.

**2.5. Transformer Encoder & Decoder Encoder** class Encoder(nn.Module):

def \_\_init\_\_(self):

super(Encoder, self).\_\_init\_\_() self.src\_emb = nn.Embedding(src\_vocab\_size, d\_model)

self.pos\_emb = nn.Embedding.from\_pretrained(get\_sinusoid\_encoding\_table(src\_len+1, d\_model),freeze=True)

self.layers = nn.ModuleList([EncoderLayer() for \_ in range(n\_layers)])

def forward(self, enc\_inputs):

enc\_outputs = self.src\_emb(enc\_inputs) + self.pos\_emb(torch.LongTensor([[1,2,3,4,0]])) enc\_self\_attn\_mask = get\_attn\_pad\_mask(enc\_inputs, enc\_inputs)

enc\_self\_attns = [] for layer in self.layers:

enc\_outputs, enc\_self\_attn = layer(enc\_outputs, enc\_self\_attn\_mask) enc\_self\_attns.append(enc\_self\_attn)

return enc\_outputs, enc\_self\_attns

• The encoder processes input embeddings through **multiple layers**. **Decoder** class Decoder(nn.Module):

def \_\_init\_\_(self):

super(Decoder, self).\_\_init\_\_() self.tgt\_emb = nn.Embedding(tgt\_vocab\_size, d\_model)

self.pos\_emb = nn.Embedding.from\_pretrained(get\_sinusoid\_encoding\_table(tgt\_len+1, d\_model),freeze=True) self.layers = nn.ModuleList([DecoderLayer() for \_ in range(n\_layers)])

def forward(self, dec\_inputs, enc\_inputs, enc\_outputs):

dec\_outputs = self.tgt\_emb(dec\_inputs) + self.pos\_emb(torch.LongTensor([[5,1,2,3,4]]))

for layer in self.layers:

dec\_outputs, dec\_self\_attn, dec\_enc\_attn = layer(dec\_outputs, enc\_outputs, dec\_self\_attn\_mask, dec\_enc\_attn\_mask) return dec\_outputs

• Similar to the encoder, but also **attends to encoder outputs**.

**3. Full Transformer Model** class Transformer(nn.Module):

def \_\_init\_\_(self):

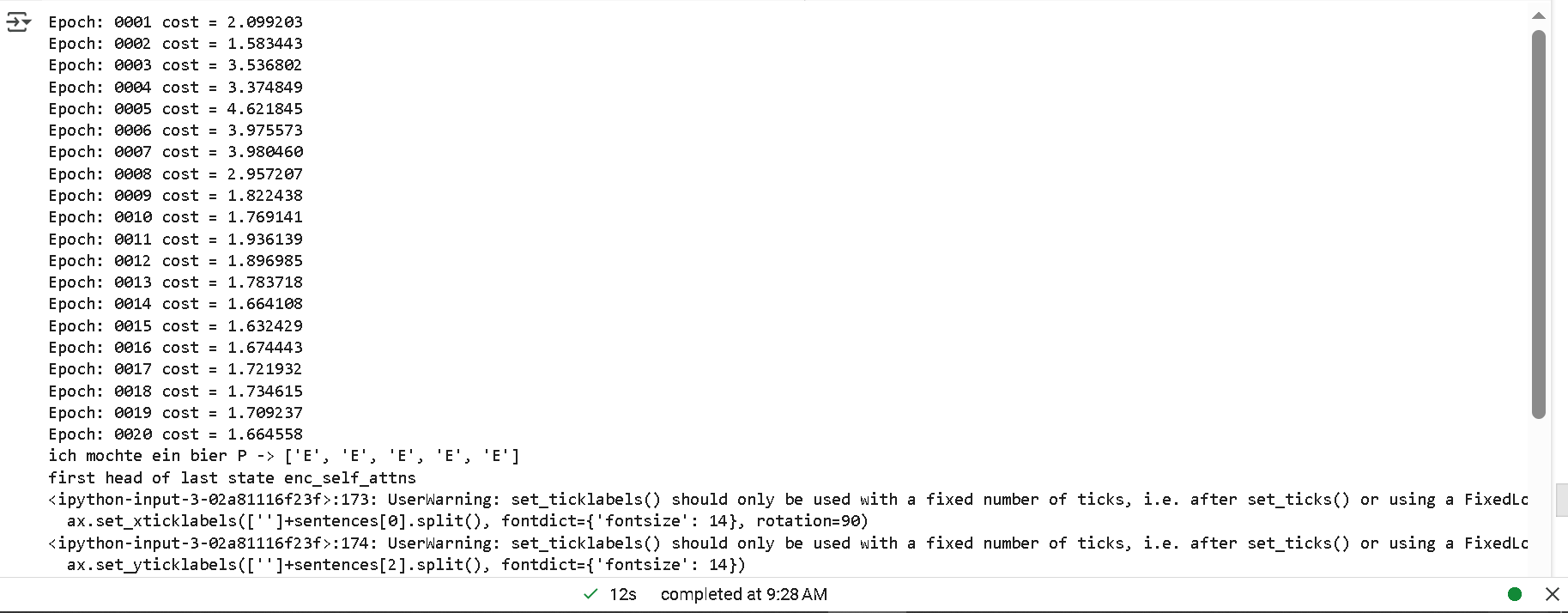
super(Transformer, self).\_\_init\_\_() self.encoder = Encoder() self.decoder = Decoder() self.projection = nn.Linear(d\_model, tgt\_vocab\_size, bias=False)

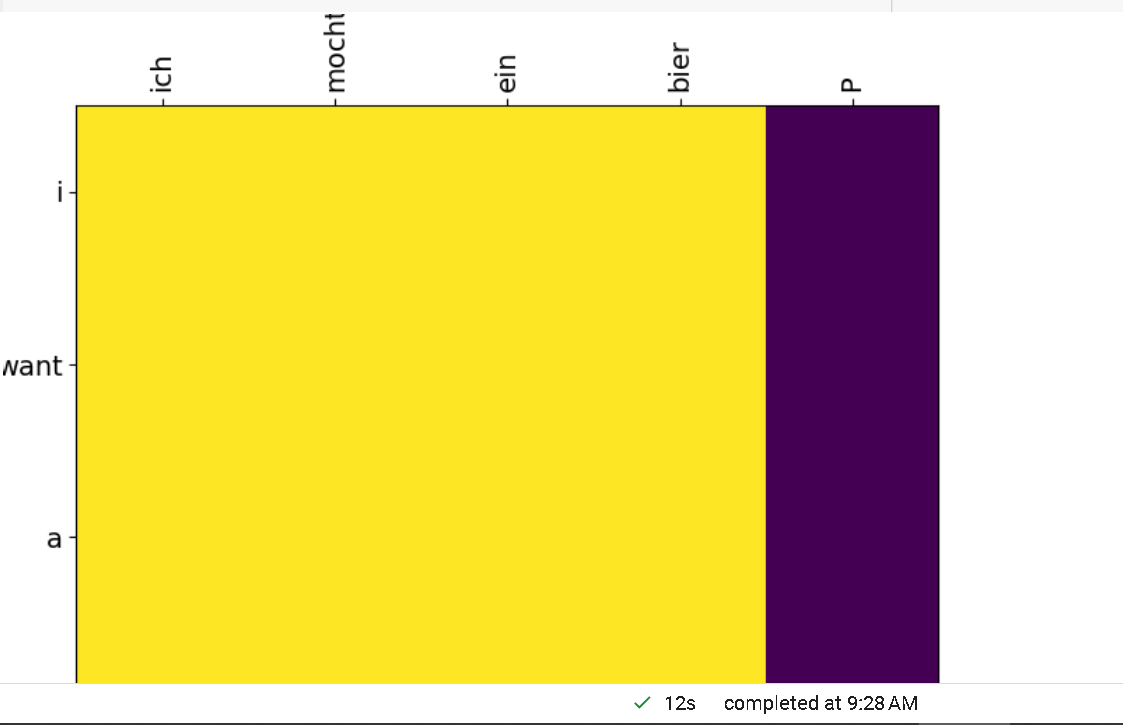
def forward(self, enc\_inputs, dec\_inputs):

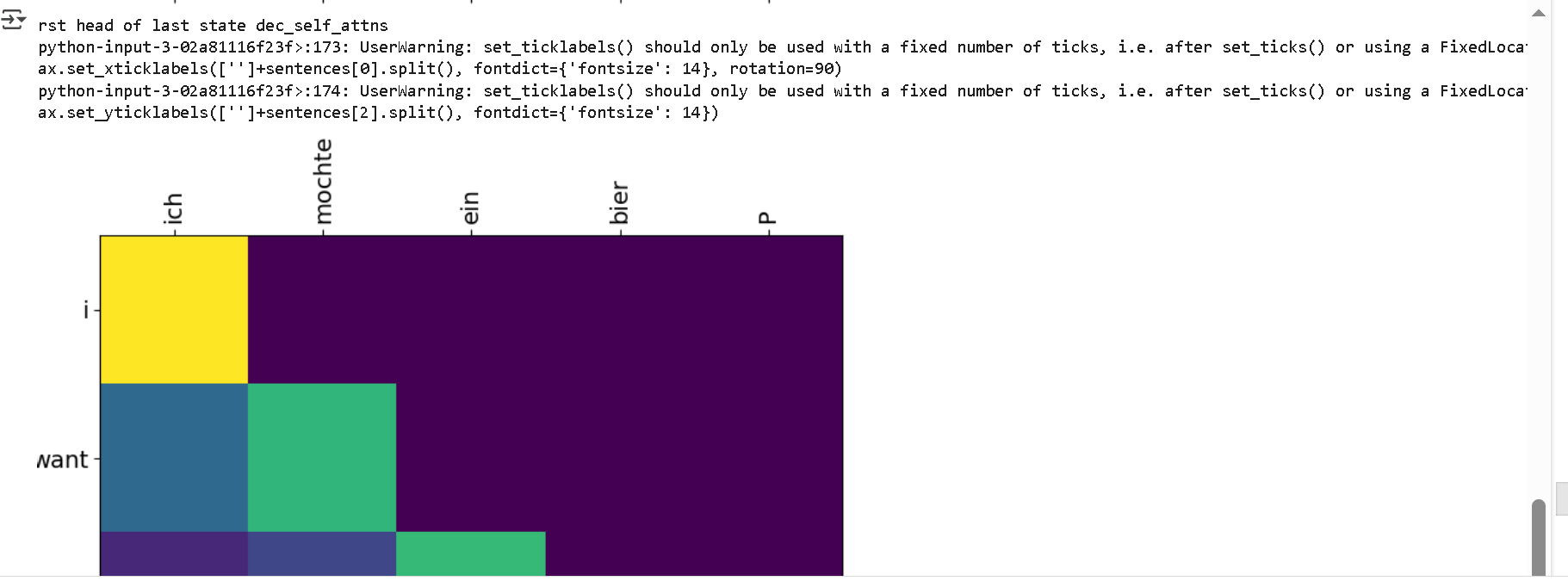
enc\_outputs, enc\_self\_attns = self.encoder(enc\_inputs) dec\_outputs, dec\_self\_attns, dec\_enc\_attns = self.decoder(dec\_inputs, enc\_inputs, enc\_outputs) dec\_logits = self.projection(dec\_outputs)

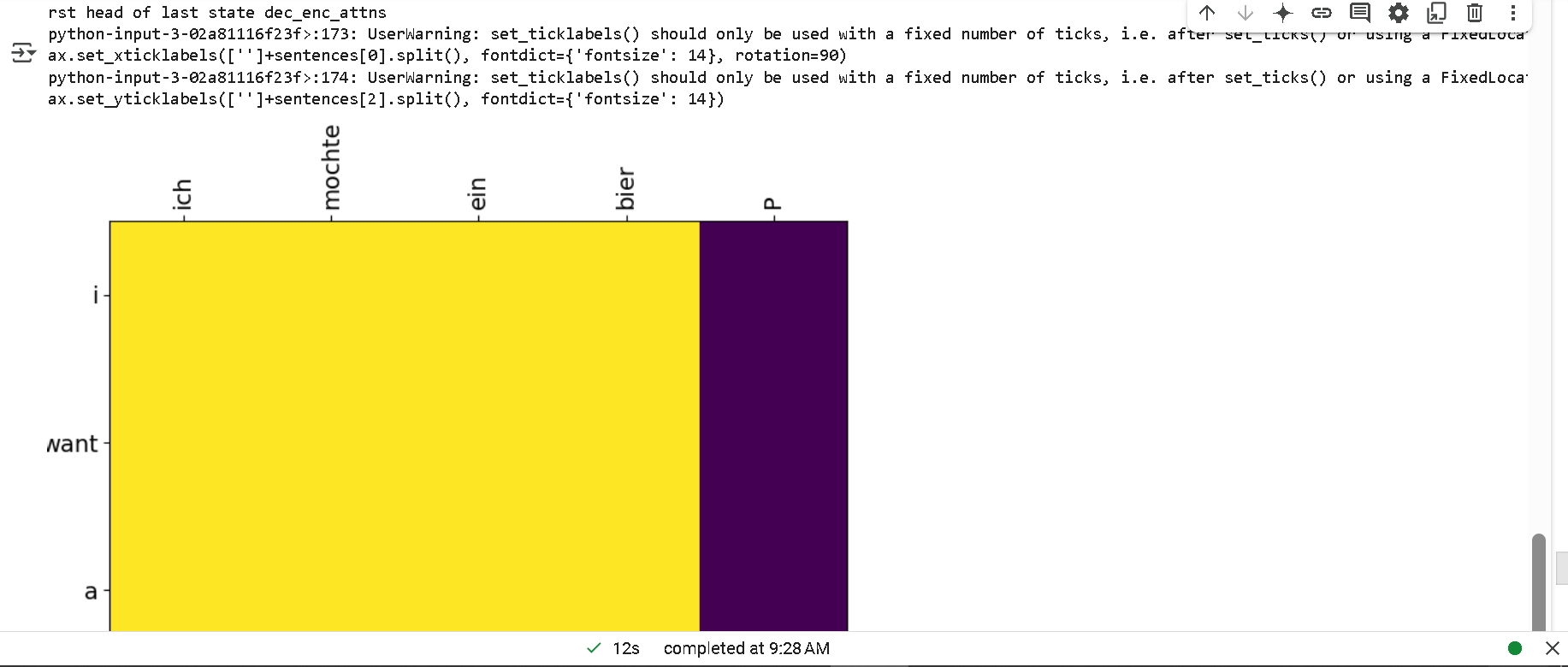
return dec\_logits.view(-1, dec\_logits.size(-1)), enc\_self\_attns, dec\_self\_attns, dec\_enc\_attns

* Combines **encoder** and **decoder** for full translation.

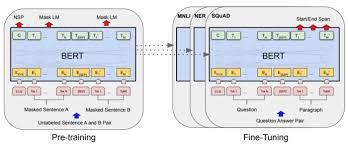
****

****

****

****

**Section 6.2: BERT – Deep Bidirectional Transformers Model Architecture Diagram:**



**Understanding the code:**

**Overview**

* The code implements a simplified version of **BERT**.
* It includes **masked language modeling (MLM)** and **next sentence prediction (NSP)**.
* Uses **Transformer encoders** with **multi-head self-attention**.
* Employs **GELU activation** and **layer normalization**.
* Prepares data by tokenizing text, applying masking, and creating input representations.

**Key Components**

1. **Data Preparation (make\_batch)** 
   * Creates input sequences with **masked tokens** and **segment IDs**.
   * Randomly selects sentence pairs for **Next Sentence Prediction (NSP)**.
   * **15% of tokens are masked** for **Masked Language Modeling (MLM)**.
   * Uses **[CLS] (classification token)** and **[SEP] (separator token)**.
2. **Embedding Layer (Embedding)** 
   * **Token embedding**: Maps words to vectors. • **Position embedding**: Captures word order.
   * **Segment embedding**: Differentiates sentence pairs.
3. **Multi-Head Attention (MultiHeadAttention)** 
   * Projects queries, keys, and values.
   * Splits them into **multiple attention heads**.
   * Uses **scaled dot-product attention**.
4. **Transformer Encoder (EncoderLayer)** 
   * **Multi-head self-attention** layer.
   * **Feedforward network (FFN)** with **GELU activation**.
5. **BERT Model (BERT)** 
   * Stacks multiple **encoder layers**.
   * Uses **softmax classification** for NSP.
   * Uses **linear projection** for MLM.

**Execution (\_\_main\_\_ block)**

* + Defines **BERT model parameters**.
  + Tokenizes and preprocesses sample text.
  + Assigns numerical representations.
  + Initializes the model.

**Summary**

This code builds a **basic BERT model** from scratch using PyTorch, including:

* + **Tokenization**
  + **Embedding**
  + **Transformer encoders**
  + **Multi-head self-attention**
  + **Masked Language Modeling (MLM)**
  + **Next Sentence Prediction (NSP)**

****