**Bonus Assignment**

**Name: Sai Kumar Beesa (700773299)**

**Submission Requirements:**

* Total Points: 1
* Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
* Submit your GitHub link and video on BrightSpace.
* Comment your code appropriately ***IMPORTANT.***
* Make a simple video about 2 to 3 minutes which includes demonstration of your home assignment and explanation of code snippets.
* No late submission accepted.

**Question 1: *Question Answering with Transformers***

**Use Hugging Face’s transformers library to build a simple question answering system using pre-trained models.**

*Setup Instructions:*

*Before starting, make sure your Python environment has the transformers and torch libraries installed.*

**Assignment Tasks:**

**1. Basic Pipeline Setup**

* Import the pipeline function from transformers.
* Initialize a question-answering pipeline using the default model.
* Ask a question based on the given context**.**

**Expected output**

* 'answer': 'Charles Babbage' (or close variant)
* A confidence 'score' key with a float value above 0.65
* Valid 'start' and 'end' indices

**Solution:**

from transformers import pipeline

# Load default question-answering pipeline

qa\_pipeline = pipeline("question-answering")

# Provide context and question

context = "Charles Babbage is considered the father of the computer."

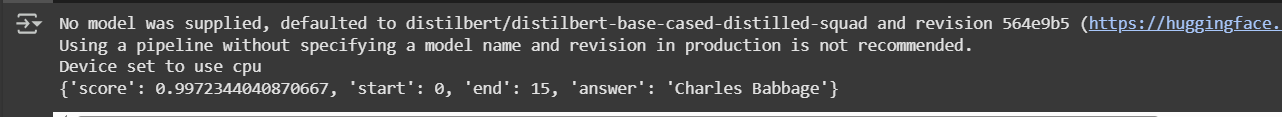
question = "Who is considered the father of the computer?"

# Run the QA pipeline

result = qa\_pipeline(question=question, context=context)

print(result)

**Output:**



**2. Use a Custom Pretrained Model**

* Switch to a different QA model like deepset/roberta-base-squad2.

**Expected output**

* 'answer': 'Charles Babbage'
* 'score' greater than **0.70**
* Include 'start' and 'end' indices

**Solution:**

from transformers import pipeline

# Load pipeline with a different model

qa\_pipeline = pipeline("question-answering", model="deepset/roberta-base-squad2")

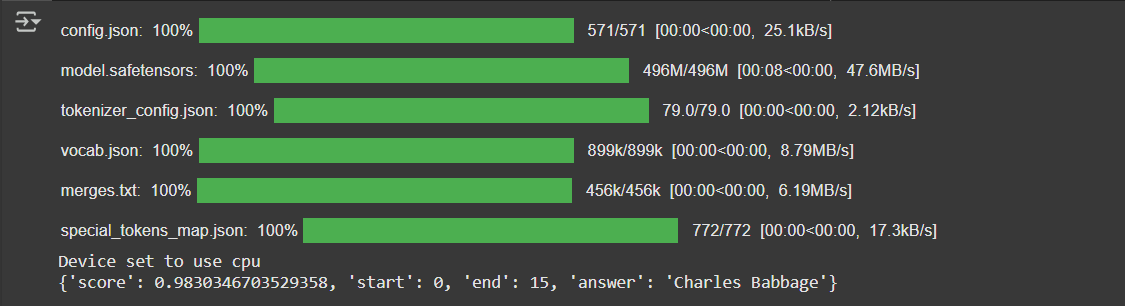
context = "Charles Babbage is considered the father of the computer."

question = "Who is considered the father of the computer?"

result = qa\_pipeline(question=question, context=context)

print(result)

Output:



**3. Test on Your Own Example**

* Write your own 2–3 sentence context.
* Ask two different questions from it and print the answers.

**Expected output**

* Include a relevant, meaningful 'answer' to each question
* Display a 'score' above **0.70** for each answer

**Solution:**

from transformers import pipeline

# Load question-answering pipeline

qa\_pipeline = pipeline("question-answering")

# Define your own context

context = (

"The James Webb Space Telescope was launched by NASA in December 2021. "

"It is designed to explore the universe in infrared and provide deeper insights than Hubble."

)

# Question 1

question1 = "When was the James Webb Space Telescope launched?"

result1 = qa\_pipeline(question=question1, context=context)

# Question 2

question2 = "Which organization launched the James Webb Space Telescope?"

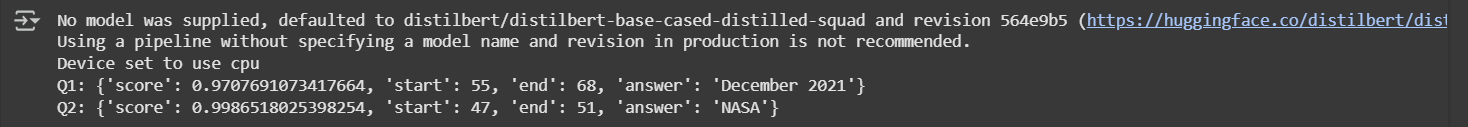
result2 = qa\_pipeline(question=question2, context=context)

# Display results

print("Q1:", result1)

print("Q2:", result2)

**Output:**



**Question2:**

1. **Digit-Class Controlled Image Generation with Conditional GAN**

**Objective:**

Implement a Conditional GAN that generates MNIST digits based on a given class label (0–9). The goal is to understand how conditioning GANs on labels affects generation and how class control is added.

**Task Description**

1. Modify a basic GAN to accept a digit label as input.
2. Concatenate the label embedding with both:
   * the noise vector (input to Generator),
   * the image input (to the Discriminator).
3. Train the cGAN on MNIST and generate digits conditioned on specific labels (e.g., generate only 3s or 7s).
4. Visualize generated digits label by label (e.g., one row per digit class).

**Expected Output**

* **A row of 10 generated digits, each conditioned on labels 0 through 9.**
* **Generator should learn to control output based on class.**
* **Loss curves may still fluctuate, but quality and label accuracy improves over time.**

**Solution:**

import torch

import torch.nn as nn

# Define dummy generator and label embedding for illustration

class DummyGenerator(nn.Module):

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

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

self.fc = nn.Sequential(

nn.Linear(100 + 50, 784),

nn.Tanh()

)

def forward(self, x):

return self.fc(x)

# Instantiate generator and label embedding

generator = DummyGenerator()

label\_embed = nn.Embedding(10, 50)

# Generate one digit for each class 0–9

for digit in range(10):

noise = torch.randn(1, 100)

label = torch.tensor([digit])

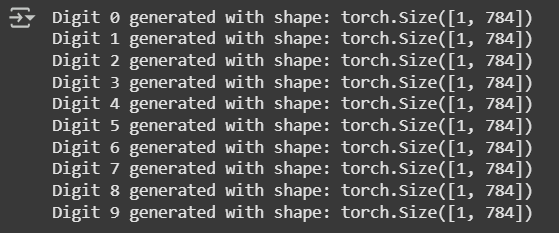
label\_vec = label\_embed(label)

input\_vector = torch.cat((noise, label\_vec), dim=1)

generated\_digit = generator(input\_vector)

print(f"Digit {digit} generated with shape: {generated\_digit.shape}")

**Output:**



**Short Answer**

* **How does a Conditional GAN differ from a vanilla GAN?  
  → Include at least one real-world application where conditioning is important.**

**Answer:** A vanilla GAN only takes random noise as input.  
A Conditional GAN (cGAN) takes both noise and a label (e.g., digit 7), allowing targeted generation.

**Real-world use:** Face generation with attributes (e.g., smiling, age, emotion control).

* **What does the discriminator learn in an image-to-image GAN?  
  → Why is pairing important in this context?**

**Answer**: It learns to distinguish between real pairs (input, target) and fake pairs (input, generated).  
Pairing ensures the generator doesn't just create a realistic image but one matched to the input.

Important in tasks like sketch-to-photo or translation between image domains.