Sentence-BERT

Reference Code

In [1]:
```
import os
import math
import re
from random import *
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "mps" if torch.backends.mps.is_available() else "cpu")
device
```

Out[1]:
```
device(type='cuda')
```

1. Data

Train, Test, Validation

In [2]:
```
import datasets

snli = datasets.load_dataset("snli")
mnli = datasets.load_dataset("glue", "mnli")
mnli["train"].features, snli["train"].features
```

Out[2]:
```
({'premise': Value(dtype='string', id=None),
  'hypothesis': Value(dtype='string', id=None),
  'label': ClassLabel(names=['entailment', 'neutral', 'contradiction'], id=None),
  'idx': Value(dtype='int32', id=None)},
 {'premise': Value(dtype='string', id=None),
  'hypothesis': Value(dtype='string', id=None),
  'label': ClassLabel(names=['entailment', 'neutral', 'contradiction'], id=None)})
```

In [3]:
```
# List of datasets to remove 'idx' column from
mnli.column_names.keys()
```

Out[3]:
```
dict_keys(['train', 'validation_matched', 'validation_mismatched', 'test_matched', 'test_mismatched'])
```

In [4]:
```
# Remove 'idx' column from each dataset
for column_names in mnli.column_names.keys():
    mnli[column_names] = mnli[column_names].remove_columns("idx")
```

In [5]:
```
mnli.column_names.keys()
```

Out[5]:
```
dict_keys(['train', 'validation_matched', 'validation_mismatched', 'test_matched', 'test_mismatched'])
```

In [6]:
```
import numpy as np
```

```python
np.unique(mnli["train"]["label"]), np.unique(snli["train"]["label"])
# snli also have -1
```

Out[6]: (array([0, 1, 2]), array([-1, 0, 1, 2]))

In [7]:
```python
# there are -1 values in the label feature, these are where no class could be decided so we remove
snli = snli.filter(lambda x: 0 if x["label"] == -1 else 1)
```

In [8]:
```python
import numpy as np

np.unique(mnli["train"]["label"]), np.unique(snli["train"]["label"])
# snli also have -1
```

Out[8]: (array([0, 1, 2]), array([0, 1, 2]))

In [9]:
```python
# Assuming you have your two DatasetDict objects named snli and mnli
from datasets import DatasetDict

# Merge the two DatasetDict objects
raw_dataset = DatasetDict(
    {
        "train": datasets.concatenate_datasets([snli["train"], mnli["train"]]).shuffle().select(list(range(100000))),
        "test": datasets.concatenate_datasets([snli["test"], mnli["test_mismatched"]]).shuffle().select(list(range(10000))),
        "validation": datasets.concatenate_datasets([snli["validation"], mnli["validation_mismatched"]]).shuffle().select(list(range(10000))),
    }
)
# remove .select(list(range(1000))) in order to use full dataset
# Now, merged_dataset_dict contains the combined datasets from snli and mnli
raw_dataset
```

Out[9]:
```
DatasetDict({
    train: Dataset({
        features: ['premise', 'hypothesis', 'label'],
        num_rows: 100000
    })
    test: Dataset({
        features: ['premise', 'hypothesis', 'label'],
        num_rows: 10000
    })
    validation: Dataset({
        features: ['premise', 'hypothesis', 'label'],
        num_rows: 10000
    })
})
```

2. Preprocessing

In [10]:
```python
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

In [11]:
```python
def preprocess_function(examples):
    max_seq_length = 128
    padding = "max_length"
    # Tokenize the premise
    premise_result = tokenizer(examples["premise"], padding=padding, max_length=max_seq_length, truncation=True)
    # num_rows, max_seq_length
```

```
    # Tokenize the hypothesis
    hypothesis_result = tokenizer(examples["hypothesis"], padding=padding, max_length=max_seq_length, truncation=True)
    # num_rows, max_seq_length
    # Extract labels
    labels = examples["label"]
    # num_rows
    return {
        "premise_input_ids": premise_result["input_ids"],
        "premise_attention_mask": premise_result["attention_mask"],
        "hypothesis_input_ids": hypothesis_result["input_ids"],
        "hypothesis_attention_mask": hypothesis_result["attention_mask"],
        "labels": labels,
    }


tokenized_datasets = raw_dataset.map(
    preprocess_function,
    batched=True,
)

tokenized_datasets = tokenized_datasets.remove_columns(["premise", "hypothesis", "label"])
tokenized_datasets.set_format("torch")
```

```
Map:   0%|          | 0/100000 [00:00<?, ? examples/s]
Map:   0%|          | 0/10000 [00:00<?, ? examples/s]
Map:   0%|          | 0/10000 [00:00<?, ? examples/s]
```

In [12]:

```
tokenized_datasets
```

Out[12]:

```
DatasetDict({
    train: Dataset({
        features: ['premise_input_ids', 'premise_attention_mask', 'hypothesis_input_ids', 'hypothesis_attention_mask', 'labels'],
        num_rows: 100000
    })
    test: Dataset({
        features: ['premise_input_ids', 'premise_attention_mask', 'hypothesis_input_ids', 'hypothesis_attention_mask', 'labels'],
        num_rows: 10000
    })
    validation: Dataset({
        features: ['premise_input_ids', 'premise_attention_mask', 'hypothesis_input_ids', 'hypothesis_attention_mask', 'labels'],
        num_rows: 10000
    })
})
```

3. Data loader

In [13]:

```
from torch.utils.data import DataLoader

# initialize the dataloader
batch_size = 32
train_dataloader = DataLoader(tokenized_datasets["train"], batch_size=batch_size, shuffle=True)
eval_dataloader = DataLoader(tokenized_datasets["validation"], batch_size=batch_size)
test_dataloader = DataLoader(tokenized_datasets["test"], batch_size=batch_size)
```

In [14]:

```
for batch in train_dataloader:
    print(batch["premise_input_ids"].shape)
    print(batch["premise_attention_mask"].shape)
    print(batch["hypothesis_input_ids"].shape)
    print(batch["hypothesis_attention_mask"].shape)
```

```python
    print(batch["labels"].shape)
    break
```

```
torch.Size([32, 128])
torch.Size([32, 128])
torch.Size([32, 128])
torch.Size([32, 128])
torch.Size([32])
```

4. Model

In [15]:

```python
# start from a pretrained bert-base-uncased model
from transformers import BertTokenizer, BertModel


model = BertModel.from_pretrained("bert-base-uncased")
model.load_state_dict(torch.load("bert_only_weights.pth", map_location=device))
model.to(device)
```

Out[15]:
```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0-11): 12 x BertLayer(
        (attention): BertAttention(
          (self): BertSdpaSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
          (intermediate_act_fn): GELUActivation()
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
```

Pooling

SBERT adds a pooling operation to the output of BERT / RoBERTa to derive a fixed sized sentence embedding

In [16]:
```python
# define mean pooling function
def mean_pool(token_embeds, attention_mask):
    # reshape attention_mask to cover 768-dimension embeddings
    in_mask = attention_mask.unsqueeze(-1).expand(token_embeds.size()).float()
    # perform mean-pooling but exclude padding tokens (specified by in_mask)
    pool = torch.sum(token_embeds * in_mask, 1) / torch.clamp(in_mask.sum(1), min=1e-9)
    return pool
```

5. Loss Function

Classification Objective Function

We concatenate the sentence embeddings u and v with the element-wise difference $|u - v|$ and multiply the result with the trainable weight $W_t \in \mathbb{R}^{3n \times k}$:

$o = \text{softmax}\left(W^T \cdot (u, v, |u - v|)\right)$

where n is the dimension of the sentence embeddings and k the number of labels. We optimize cross-entropy loss. This structure is depicted in Figure 1.

Regression Objective Function.

The cosine similarity between the two sentence embeddings u and v is computed (Figure 2). We use means quared-error loss as the objective function.

(Manhatten / Euclidean distance, semantically similar sentences can be found.)

In [17]:
```
def configurations(u, v):
    # build the |u-v| tensor
    uv = torch.sub(u, v)  # batch_size,hidden_dim
    uv_abs = torch.abs(uv)  # batch_size,hidden_dim

    # concatenate u, v, |u-v|
    x = torch.cat([u, v, uv_abs], dim=-1)  # batch_size, 3*hidden_dim
    return x


def cosine_similarity(u, v):
    dot_product = np.dot(u, v)
    norm_u = np.linalg.norm(u)
    norm_v = np.linalg.norm(v)
    similarity = dot_product / (norm_u * norm_v)
    return similarity
```

In [18]:
```
classifier_head = torch.nn.Linear(768 * 3, 3).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
optimizer_classifier = torch.optim.Adam(classifier_head.parameters(), lr=2e-5)

criterion = nn.CrossEntropyLoss()
```

In [19]:
```
from transformers import get_linear_schedule_with_warmup

# and setup a warmup for the first ~10% steps
total_steps = int(len(raw_dataset) / batch_size)
warmup_steps = int(0.1 * total_steps)
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=total_steps - warmup_steps)

# then during the training loop we update the scheduler per step
scheduler.step()

scheduler_classifier = get_linear_schedule_with_warmup(
    optimizer_classifier, num_warmup_steps=warmup_steps, num_training_steps=total_steps - warmup_steps
)

# then during the training loop we update the scheduler per step
scheduler_classifier.step()
```

```
c:\Users\silan\Desktop\A4\.venv\lib\site-packages\torch\optim\lr_scheduler.py:143: UserWarning: Detected call of `lr_scheduler.step()` before `optimizer.step()`. In PyTorch 1.1.0 and later, yo
u should call them in the opposite order: `optimizer.step()` before `lr_scheduler.step()`.  Failure to do this will result in PyTorch skipping the first value of the learning rate schedule. Se
e more details at https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate
  warnings.warn("Detected call of `lr_scheduler.step()` before `optimizer.step()`. "
```

6. Training

In [20]:

```python
from tqdm.auto import tqdm

torch.cuda.empty_cache()

num_epoch = 5
# 1 epoch should be enough, increase if wanted
for epoch in range(num_epoch):
    model.train()
    classifier_head.train()
    # initialize the dataloader loop with tqdm (tqdm == progress bar)
    for step, batch in enumerate(tqdm(train_dataloader, leave=True)):
        # zero all gradients on each new step
        optimizer.zero_grad()
        optimizer_classifier.zero_grad()

        # prepare batches and more all to the active device
        inputs_ids_a = batch["premise_input_ids"].to(device)
        inputs_ids_b = batch["hypothesis_input_ids"].to(device)
        attention_a = batch["premise_attention_mask"].to(device)
        attention_b = batch["hypothesis_attention_mask"].to(device)
        label = batch["labels"].to(device)

        # extract token embeddings from BERT at last_hidden_state
        u = model(inputs_ids_a, attention_mask=attention_a)
        v = model(inputs_ids_b, attention_mask=attention_b)

        u_last_hidden_state = u.last_hidden_state  # all token embeddings A = batch_size, seq_len, hidden_dim
        v_last_hidden_state = v.last_hidden_state  # all token embeddings B = batch_size, seq_len, hidden_dim

        # get the mean pooled vectors
        u_mean_pool = mean_pool(u_last_hidden_state, attention_a)  # batch_size, hidden_dim
        v_mean_pool = mean_pool(v_last_hidden_state, attention_b)  # batch_size, hidden_dim

        # build the |u-v| tensor
        uv = torch.sub(u_mean_pool, v_mean_pool)  # batch_size,hidden_dim
        uv_abs = torch.abs(uv)  # batch_size,hidden_dim

        # concatenate u, v, |u-v|
        x = torch.cat([u_mean_pool, v_mean_pool, uv_abs], dim=-1)  # batch_size, 3*hidden_dim

        # process concatenated tensor through classifier_head
        x = classifier_head(x)  # batch_size, classifer

        # calculate the 'softmax-loss' between predicted and true label
        loss = criterion(x, label)

        # using loss, calculate gradients and then optimizerize
        loss.backward()
        optimizer.step()
        optimizer_classifier.step()
```

```
        scheduler.step()  # update learning rate scheduler
        scheduler_classifier.step()

    print(f"Epoch: {epoch + 1} | loss = {loss.item():.6f}")
```

```
  0%|          | 0/3125 [00:00<?, ?it/s]
c:\Users\silan\Desktop\A4\.venv\lib\site-packages\transformers\models\bert\modeling_bert.py:440: UserWarning: 1Torch was not compiled with flash attention. (Triggered internally at C:\actions-runner\_work\pytorch\pytorch\builder\windows\pytorch\aten\src\ATen\native\transformers\cuda\sdp_utils.cpp:263.)
  attn_output = torch.nn.functional.scaled_dot_product_attention(
Epoch: 1 | loss = 1.069691
  0%|          | 0/3125 [00:00<?, ?it/s]
Epoch: 2 | loss = 1.135245
  0%|          | 0/3125 [00:00<?, ?it/s]
Epoch: 3 | loss = 1.097306
  0%|          | 0/3125 [00:00<?, ?it/s]
Epoch: 4 | loss = 1.119025
  0%|          | 0/3125 [00:00<?, ?it/s]
Epoch: 5 | loss = 1.165591
```

In [21]:

```
from sklearn.metrics import classification_report

torch.cuda.empty_cache()

model.eval()
classifier_head.eval()
all_preds = []
all_labels = []

with torch.no_grad():
    for step, batch in enumerate(eval_dataloader):
        # prepare batches and move all to the active device
        inputs_ids_a = batch["premise_input_ids"].to(device)
        inputs_ids_b = batch["hypothesis_input_ids"].to(device)
        attention_a = batch["premise_attention_mask"].to(device)
        attention_b = batch["hypothesis_attention_mask"].to(device)
        labels = batch["labels"].to(device)

        # extract token embeddings from BERT at last_hidden_state
        u = model(inputs_ids_a, attention_mask=attention_a)[0]  # all token embeddings A = batch_size, seq_len, hidden_dim
        v = model(inputs_ids_b, attention_mask=attention_b)[0]  # all token embeddings B = batch_size, seq_len, hidden_dim

        # get the mean pooled vectors
        u_mean_pool = mean_pool(u, attention_a)  # batch_size, hidden_dim
        v_mean_pool = mean_pool(v, attention_b)  # batch_size, hidden_dim

        # build the |u-v| tensor
        uv = torch.sub(u_mean_pool, v_mean_pool)  # batch_size,hidden_dim
        uv_abs = torch.abs(uv)  # batch_size,hidden_dim

        # concatenate u, v, |u-v|
        x = torch.cat([u_mean_pool, v_mean_pool, uv_abs], dim=-1)  # batch_size, 3*hidden_dim

        # process concatenated tensor through classifier_head
        logits = classifier_head(x)  # batch_size, classifer

        # get predictions
        preds = torch.argmax(logits, dim=-1)

        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
```

```python
# Print classification report
print(classification_report(all_labels, all_preds, target_names=["entailment", "neutral", "contradiction"]))
```

```
               precision    recall  f1-score   support

   entailment       0.42      0.02      0.05      3486
      neutral       0.33      0.75      0.46      3199
contradiction       0.33      0.25      0.28      3315

     accuracy                           0.33     10000
    macro avg       0.36      0.34      0.26     10000
 weighted avg       0.36      0.33      0.26     10000
```

7. Inference

In [22]:

```python
import torch
from sklearn.metrics.pairwise import cosine_similarity


def calculate_similarity(model, tokenizer, sentence_a, sentence_b, device):
    # Tokenize and convert sentences to input IDs and attention masks
    inputs_a = tokenizer(sentence_a, return_tensors="pt", truncation=True, padding=True).to(device)
    inputs_b = tokenizer(sentence_b, return_tensors="pt", truncation=True, padding=True).to(device)

    # Move input IDs and attention masks to the active device
    inputs_ids_a = inputs_a["input_ids"]
    attention_a = inputs_a["attention_mask"]
    inputs_ids_b = inputs_b["input_ids"]
    attention_b = inputs_b["attention_mask"]

    # Extract token embeddings from BERT
    u = model(inputs_ids_a, attention_mask=attention_a)[0]  # all token embeddings A = batch_size, seq_len, hidden_dim
    v = model(inputs_ids_b, attention_mask=attention_b)[0]  # all token embeddings B = batch_size, seq_len, hidden_dim

    # Get the mean-pooled vectors
    u = mean_pool(u, attention_a).detach().cpu().numpy().reshape(-1)  # batch_size, hidden_dim
    v = mean_pool(v, attention_b).detach().cpu().numpy().reshape(-1)  # batch_size, hidden_dim

    # Calculate cosine similarity
    similarity_score = cosine_similarity(u.reshape(1, -1), v.reshape(1, -1))[0, 0]

    return similarity_score


# Example usage:
sentence_a = "Your contribution helped make it possible for us to provide our students with a quality education."
sentence_b = "Your contributions were of no help with our students' education."
similarity = calculate_similarity(model, tokenizer, sentence_a, sentence_b, device)
print(f"Cosine Similarity: {similarity:.4f}")
```

```
Cosine Similarity: 0.8248
```

In [23]:

```python
sentences = [
    # Entailment pairs
    ("A man is playing guitar on stage.", "A person is performing music."),
    ("She is cooking dinner in the kitchen.", "A woman is preparing a meal."),
    ("The children are playing in the park.", "Kids are having fun outdoors."),
```

```python
    ("He is reading a book quietly.", "A man is enjoying a book."),
    # Neutral pairs
    ("The sun is shining brightly.", "I am planning to go for a walk."),
    ("She bought a new dress.", "The store had a big sale yesterday."),
    ("The car is parked outside.", "It might rain later in the evening."),
    ("They are watching a movie.", "The theater was crowded last night."),
    # Contradiction pairs
    ("The dog is barking loudly.", "The neighborhood is completely silent."),
    ("He passed the exam easily.", "He failed all his tests this semester."),
]

for sentence_a, sentence_b in sentences:
    similarity = calculate_similarity(model, tokenizer, sentence_a, sentence_b, device)
    print(f"Sentence A: {sentence_a}")
    print(f"Sentence B: {sentence_b}")
    print(f"Cosine Similarity: {similarity:.4f}\n")
```

```
Sentence A: A man is playing guitar on stage.
Sentence B: A person is performing music.
Cosine Similarity: 0.7259


Sentence B: A person is performing music.
Cosine Similarity: 0.7259

Sentence A: She is cooking dinner in the kitchen.
Sentence B: A woman is preparing a meal.
Cosine Similarity: 0.6535

Sentence A: The children are playing in the park.
Sentence B: Kids are having fun outdoors.
Cosine Similarity: 0.6873

Sentence A: He is reading a book quietly.
Sentence B: A man is enjoying a book.
Cosine Similarity: 0.6883

Sentence A: The sun is shining brightly.
Sentence B: I am planning to go for a walk.
Cosine Similarity: 0.4353

Sentence A: She bought a new dress.
Sentence B: The store had a big sale yesterday.
Cosine Similarity: 0.5045

Sentence A: The car is parked outside.
Sentence B: It might rain later in the evening.
Cosine Similarity: 0.4254

Sentence A: They are watching a movie.
Sentence B: The theater was crowded last night.
Cosine Similarity: 0.3582

Sentence A: The dog is barking loudly.
Sentence B: The neighborhood is completely silent.
Cosine Similarity: 0.6425

Sentence A: He passed the exam easily.
Sentence B: He failed all his tests this semester.
Cosine Similarity: 0.6576
```

In [24]:

```python
# Randomly pick 5 pairs of sentences from snli with different entailment relationships
import random

snli_entailment = snli["validation"].filter(lambda x: x["label"] == 0)
snli_neutral = snli["validation"].filter(lambda x: x["label"] == 1)
snli_contradiction = snli["validation"].filter(lambda x: x["label"] == 2)

random_entailment = random.sample(list(snli_entailment), 5)
random_neutral = random.sample(list(snli_neutral), 5)
random_contradiction = random.sample(list(snli_contradiction), 5)

print("Entailment Examples:")
for example in random_entailment:
    sentence_a = example["premise"]
    sentence_b = example["hypothesis"]
```

```python
    similarity = calculate_similarity(model, tokenizer, sentence_a, sentence_b, device)
    # print(f"Sentence A: {sentence_a}")
    # print(f"Sentence B: {sentence_b}")
    print(f"Cosine Similarity: {similarity:.4f}")

print("\nNeutral Examples:")
for example in random_neutral:
    sentence_a = example["premise"]
    sentence_b = example["hypothesis"]
    similarity = calculate_similarity(model, tokenizer, sentence_a, sentence_b, device)
    # print(f"Sentence A: {sentence_a}")
    # print(f"Sentence B: {sentence_b}")
    print(f"Cosine Similarity: {similarity:.4f}")

print("\nContradiction Examples:")
for example in random_contradiction:
    sentence_a = example["premise"]
    sentence_b = example["hypothesis"]
    similarity = calculate_similarity(model, tokenizer, sentence_a, sentence_b, device)
    # print(f"Sentence A: {sentence_a}")
    # print(f"Sentence B: {sentence_b}")
    print(f"Cosine Similarity: {similarity:.4f}")
```

```
Entailment Examples:
Cosine Similarity: 0.8104
Cosine Similarity: 0.8615
Cosine Similarity: 0.8124
Cosine Similarity: 0.7727
Cosine Similarity: 0.9073

Neutral Examples:
Cosine Similarity: 0.7053
Cosine Similarity: 0.5814
Cosine Similarity: 0.8302
Cosine Similarity: 0.9373
Cosine Similarity: 0.7901

Contradiction Examples:
Cosine Similarity: 0.8581
Cosine Similarity: 0.3708
Cosine Similarity: 0.4233
Cosine Similarity: 0.5587
Cosine Similarity: 0.5595
```

In [25]:

```python
entailment_similarities = [calculate_similarity(model, tokenizer, example["premise"], example["hypothesis"], device) for example in snli_entailment]
neutral_similarities = [calculate_similarity(model, tokenizer, example["premise"], example["hypothesis"], device) for example in snli_neutral]
contradiction_similarities = [
    calculate_similarity(model, tokenizer, example["premise"], example["hypothesis"], device) for example in snli_contradiction
]


print("Entailment:", min(entailment_similarities), max(entailment_similarities), sum(entailment_similarities) / len(entailment_similarities))
print("Neutral:", min(neutral_similarities), max(neutral_similarities), sum(neutral_similarities) / len(neutral_similarities))
print(
    "Contradiction:",
    min(contradiction_similarities),
    max(contradiction_similarities),
    sum(contradiction_similarities) / len(contradiction_similarities),
)
```

```
Entailment: 0.11354333 1.0 0.7228040125105752
Neutral: 0.09101771 0.9933052 0.6847346374664277
Contradiction: 0.12401185 0.9995548 0.6427295705584254
```

In [26]:
```
# Save Fine-Tuned Model
torch.save(model.state_dict(), "sbert_finetuned.pth")
torch.save(classifier_head.state_dict(), "classifier_head.pth")

print("Fine-Tuned Model Saved Successfully.")
```

```
Fine-Tuned Model Saved Successfully.
```