Assignment @ SuperAGI

# Q/A -

1. Wnew +W new (n+1) ≈ wn matlab weights wnew and wnew(n+1) will adjust in such a way that their combined effect is similar to the effect of regional weight wn.
2. E is better than A with over 95% confidence, B is worse than A with over 95% confidence. You need to run the test for longer to tell where C and D compare to A with 95% confidence.
3. O(mk) -> computational cost for each gradient descent iteration in logistic regression , when dealing with sparse data in an optimized package.
4. Approach 3 has the potential to be the most beneficial for improving the accuracy of the V2 classifier, as it focuses on challenging cases where the V1 model was confidently wrong and far from the decision boundary.

Approach 1 could also provide useful data, but the proximity to the decision boundary doesn't guarantee that the V1 classifier was wrong.

Approach 2, while providing labeled data, may not specifically target challenging cases where the classifier is likely to make errors.

**Ranking (from likely most helpful to least):**

1. Approach 3
2. Approach 1
3. Approach 2
4. MLE -> k/n , Bayesian : K+1/n+2 , map estimate : k+1/n+2

# Coding

All these 3 tasks are done with help of chatgpt.

## Task 1

The following python code is as follows-

import torch

import torch.nn as nn

class GPT2Model(nn.Module):

def \_\_init\_\_(self, vocab\_size, d\_model=768, nhead=12, num\_layers=12):

super(GPT2Model, self).\_\_init\_\_()

self.token\_embedding = nn.Embedding(vocab\_size, d\_model)

self.positional\_encoding = self.positional\_encoding(d\_model)

self.transformer\_layers = nn.ModuleList([

nn.TransformerEncoderLayer(d\_model, nhead) for \_ in range(num\_layers)

])

self.fc = nn.Linear(d\_model, vocab\_size)

def forward(self, x):

x = self.token\_embedding(x)

x += self.positional\_encoding[:x.size(1), :].unsqueeze(0)

for layer in self.transformer\_layers:

x = layer(x)

x = self.fc(x)

return x

def positional\_encoding(self, d\_model, max\_len=512):

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* -(torch.log(torch.tensor(10000.0)) / d\_model))

pe[:, 0::2] = torch.sin(position \* div\_term)

pe[:, 1::2] = torch.cos(position \* div\_term)

return pe

vocab\_size = 30000 # Adjust based on your dataset

model = GPT2Model(vocab\_size)

checkpoint = torch.load("path/to/gpt2\_model\_checkpoint.pth")

model.load\_state\_dict(checkpoint)

model.eval()

input\_sequence = torch.randint(0, vocab\_size, (1, 50)) # Adjust sequence length as needed

output = model(input\_sequence)

print(output.shape)

## Task 2

### **Rotatory positional embedding**

import torch

import torch.nn as nn

import torch.nn.functional as F

class GPT2ModelRotary(nn.Module):

def \_\_init\_\_(self, vocab\_size, d\_model=768, nhead=12, num\_layers=12):

super(GPT2ModelRotary, self).\_\_init\_\_()

self.token\_embedding = nn.Embedding(vocab\_size, d\_model)

self.rotary\_positional\_encoding = self.rotary\_positional\_encoding(d\_model)

self.transformer\_layers = nn.ModuleList([

nn.TransformerEncoderLayer(d\_model, nhead) for \_ in range(num\_layers)

])

self.fc = nn.Linear(d\_model, vocab\_size)

def forward(self, x):

x = self.token\_embedding(x)

x += self.rotary\_positional\_encoding[:x.size(1), :].unsqueeze(0)

for layer in self.transformer\_layers:

x = layer(x)

x = self.fc(x)

return x

def rotary\_positional\_encoding(self, d\_model, max\_len=512):

pe = torch.zeros(max\_len, d\_model)

position = torch.arange(0, max\_len, dtype=torch.float).unsqueeze(1)

div\_term = torch.exp(torch.arange(0, d\_model, 2).float() \* -(torch.log(torch.tensor(10000.0)) / d\_model))

sin\_vals = torch.sin(position \* div\_term)

cos\_vals = torch.cos(position \* div\_term)

return torch.cat([sin\_vals, cos\_vals], dim=1)

model\_rotary = GPT2ModelRotary(vocab\_size)

output\_rotary = model\_rotary(input\_sequence)

print(output\_rotary.shape)

### 

### **Group Query Attention**

class GPT2ModelGroupQueryAttention(GPT2ModelRotary):

def \_\_init\_\_(self, vocab\_size, d\_model=768, nhead=12, num\_layers=12, num\_groups=4):

super(GPT2ModelGroupQueryAttention, self).\_\_init\_\_(vocab\_size, d\_model, nhead, num\_layers)

self.num\_groups = num\_groups

for layer in self.transformer\_layers:

layer.self\_attn = nn.MultiheadAttention(d\_model, nhead, kdim=d\_model // num\_groups, vdim=d\_model // num\_groups)

model\_group\_query\_attention = GPT2ModelGroupQueryAttention(vocab\_size)

output\_group\_query\_attention = model\_group\_query\_attention(input\_sequence)

print(output\_group\_query\_attention.shape)

### **Sliding Window Attention**

class GPT2ModelSlidingWindow(GPT2ModelRotary):

def \_\_init\_\_(self, vocab\_size, d\_model=768, nhead=12, num\_layers=12, window\_size=512):

super(GPT2ModelSlidingWindow, self).\_\_init\_\_(vocab\_size, d\_model, nhead, num\_layers)

self.window\_size = window\_size

for layer in self.transformer\_layers:

layer.self\_attn = SlidingWindowAttention(d\_model, nhead, window\_size)

class SlidingWindowAttention(nn.Module):

def \_\_init\_\_(self, embed\_dim, num\_heads, window\_size):

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

self.window\_size = window\_size

self.attention = nn.MultiheadAttention(embed\_dim, num\_heads)

def forward(self, x):

return x

model\_sliding\_window = GPT2ModelSlidingWindow(vocab\_size)

output\_sliding\_window = model\_sliding\_window(input\_sequence)

print(output\_sliding\_window.shape)

## Task 3

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, random\_split

class MyDataset(torch.utils.data.Dataset):

vocab\_size = 30000 # Adjust based on your dataset

model = GPT2Model(vocab\_size)

batch\_size = 64

learning\_rate = 1e-3

epochs = 10

dataset = MyDataset()

train\_size = int(0.8 \* len(dataset))

val\_size = len(dataset) - train\_size

train\_dataset, val\_dataset = random\_split(dataset, [train\_size, val\_size])

train\_loader = DataLoader(train\_dataset, batch\_size=batch\_size, shuffle=True)

val\_loader = DataLoader(val\_dataset, batch\_size=batch\_size, shuffle=False)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=learning\_rate)

def train\_step(model, data, criterion, optimizer):

model.train()

inputs, targets = data

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, targets)

loss.backward()

optimizer.step()

return loss.item()

def validate(model, val\_loader, criterion):

model.eval()

total\_loss = 0.0

with torch.no\_grad():

for inputs, targets in val\_loader:

outputs = model(inputs)

loss = criterion(outputs, targets)

total\_loss += loss.item()

return total\_loss / len(val\_loader)

# Single GPU Training Loop

for epoch in range(epochs):

for i, data in enumerate(train\_loader):

loss = train\_step(model, data, criterion, optimizer)

print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train\_loader)}, Loss: {loss}")

# Distributed Data Parallel (DDP) Training Loop

if torch.cuda.device\_count() > 1:

model = nn.DataParallel(model)

model = model.to("cuda")

model = nn.parallel.DistributedDataParallel(model)

for epoch in range(epochs):

for i, data in enumerate(train\_loader):

loss = train\_step(model, data, criterion, optimizer)

print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train\_loader)}, Loss: {loss}")

# Fully Sharded Data Parallel (FSDP) Training Loop

if torch.cuda.device\_count() > 1:

model = nn.DataParallel(model)

from fairscale.nn.data\_parallel import FullyShardedDataParallel

model = model.to("cuda")

model = FullyShardedDataParallel(model)

for epoch in range(epochs):

for i, data in enumerate(train\_loader):

loss = train\_step(model, data, criterion, optimizer)

print(f"Epoch {epoch + 1}, Batch {i + 1}/{len(train\_loader)}, Loss: {loss}")