Bert

1 - Import Data

Import all data and merge it in one dataframe

In [ ]:
```
import json

emotion = pd.read_csv('/kaggle/input/dm-2024-isa-5810-lab-2-homework/emotior
data_identification = pd.read_csv('/kaggle/input/dm-2024-isa-5810-lab-2-home

# import and convert tweet (a bit slow, could be optimized)
with open('/kaggle/input/dm-2024-isa-5810-lab-2-homework/tweets_DM.json', 'r
        data = [json.loads(line) for line in f]

df = pd.DataFrame(data)
_source = df['_source'].apply(lambda x: x['tweet'])
df = pd.DataFrame({
    'tweet_id': _source.apply(lambda x: x['tweet_id']),
    'text': _source.apply(lambda x: x['text']),
})
df = df.merge(data_identification, on='tweet_id', how='left')
```

Separate Train and test/output data

In [ ]:
```
# train test split
test_data = df[df['identification'] == 'test']

train_data = df[df['identification'] == 'train']
train_data = train_data.merge(emotion, on='tweet_id', how='left')
```

2. Preprocessing

Train validation split

In [ ]:
```
from sklearn.model_selection import train_test_split

RANDOM_STATE = 0 # ATTENTION: NON-DETERMINISTIC
TEST_SPLIT = 0.15
X_train, X_val, y_train, y_val = train_test_split(train_data['text'],
                                                    train_data['emotion'],
                                                    test_size = TEST_SPLIT,
                                                    random_state = RANDOM_ST
                                                    stratify = train_data['e
```

Encode emotions

```python
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_train = le.fit_transform(y_train)
y_val = le.transform(y_val)
```

Compute class weights

The dataset is somewhat imbalanced when it comes to the number of observations for each class, as such I decided to use class weights so that the models takes this into consideration.

Using class weights with BERT helps avoid overfitting and keeps training efficient, whereas oversampling adds redundancy, is slower, and could harm the model's ability to generalize.

```python
import torch
from sklearn.utils.class_weight import compute_class_weight
from transformers import AutoModelForSequenceClassification
from torch.nn.parallel import DistributedDataParallel as DDP

# use class weights as it is less expensive than oversampling
class_weights = compute_class_weight('balanced', classes=np.unique(y_train),
class_weights_tensor = torch.tensor(class_weights, dtype=torch.float)

device = torch.device('cuda')
class_weights_tensor = class_weights_tensor.to(device)
```

Initialize model

Initialization of BERTweet, a pre-trained model for english tweets.

```python
num_labels = len(set(y_train))
model = AutoModelForSequenceClassification.from_pretrained('vinai/bertweet-b
                                                            num_labels=num_la
                                                           )

model = model.to(device)

# Allow for dual GPU use
#model = DDP(model) is recommended but is not working
model = torch.nn.DataParallel(model)
```

Create PyTorch Dataset and Tokenize

```python
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer


BATCH_SIZE = 128 # consider reducing in case of OOM
NUM_WORKERS = 4 # 4 cores available in kaggle vm
```

```
class PostDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

        # tokenize the entire dataset upfront
        self.encodings = tokenizer(
            texts.tolist(),
            max_length=max_len,
            padding='max_length',
            truncation=True,
            return_tensors="pt",
            return_token_type_ids=False,
        )

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        text = str(self.texts.iloc[idx])
        label = self.labels[idx]

        return {
            'input_ids': self.encodings['input_ids'][idx].squeeze(0),
            'attention_mask': self.encodings['attention_mask'][idx].squeeze(
            'labels': torch.tensor(self.labels[idx], dtype=torch.long)
        }



tokenizer = AutoTokenizer.from_pretrained('vinai/bertweet-base', normalizati
max_len = 128 # max length of bertweet



train_loader = DataLoader(
        PostDataset(pd.Series(X_train), y_train, tokenizer, max_len),
        batch_size = BATCH_SIZE, shuffle=True, num_workers = NUM_WORKERS, pi
    )
test_loader = DataLoader(
        PostDataset(pd.Series(X_val), y_val, tokenizer, max_len),
        batch_size = BATCH_SIZE, num_workers = NUM_WORKERS, pin_memory=True
    )
```

3 - Training

3.1 Training Loop

In [ ]:
```
from torch.optim import AdamW
from transformers import get_scheduler
```

```
from torch.cuda.amp import autocast
from torch.amp import GradScaler

EPOCHS = 1 # Used in the final run because of time constraints, however loss

criterion = torch.nn.CrossEntropyLoss(weight=class_weights_tensor, reduction
optimizer = AdamW(model.parameters(), lr=1e-4, eps=1e-8, weight_decay=0.01)
scaler = GradScaler()

num_training_steps = len(train_loader) * EPOCHS
# Will increase the LR during warmup and then decrease it as it gets "more t
scheduler = get_scheduler(name="linear",
                          optimizer=optimizer,
                          num_warmup_steps=int(0.1 * num_training_steps),
                          num_training_steps=num_training_steps
                         )

def train_epoch(model, data_loader, criterion, optimizer, scaler, device, sc
    # Set the model to training mode
    model.train()
    running_loss = 0.0

    num_batches = len(data_loader)

    # Loop through the DataLoader
    for batch in data_loader:
        # load onto GPU
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(torch.long).to(device) # Make sure it's

        # Zero the gradients from the previous step
        optimizer.zero_grad()

        # Forward pass with autocast for mixed precision
        with autocast(device_type='cuda'):
            outputs = model(input_ids, attention_mask=attention_mask, labels
            loss = outputs.loss

        # avoid 2d error
        if loss.dim() > 0:
            loss = loss.mean()

        # backpropagation
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()
        running_loss += loss.item()

    average_loss = running_loss / num_batches
    return average_loss
```

In [ ]:

```
import torch
from torch.amp import autocast
from sklearn.metrics import f1_score
```

```python
def eval_model(model, data_loader, criterion, device):

    model.eval()
    running_loss = 0.0
    all_preds = []
    all_labels = []

    # Disable gradient calculations during evaluation for efficiency
    with torch.no_grad():
        for batch in data_loader:
            # load onto GPU
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(torch.long).to(device)  # Ensure lab

            # Forward pass with autocast for mixed precision
            with autocast(device_type='cuda'):
                outputs = model(input_ids, attention_mask=attention_mask, la
                loss = outputs.loss

            # avoid 2d error
            if loss.dim() > 0:
                loss = loss.mean()

            running_loss += loss

            # Predict
            preds = torch.argmax(outputs.logits, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Calculate the average loss
    average_loss = running_loss / len(data_loader)

    # Calculate F1 score (weighted average)
    f1 = f1_score(all_labels, all_preds, average='weighted')

    return average_loss, f1
```

In [ ]:

```python
patience = 2
best_val_loss = float('inf')
early_stop_counter = 0

for epoch in range(EPOCHS):

    # Train and validate
    train_loss = train_epoch(model, train_loader, criterion, optimizer, scal
    test_loss, f1 = eval_model(model, test_loader, criterion, device)

    # Save best model and stop early to prevent overfitting
    if test_loss < best_val_loss:
        best_val_loss = test_loss
        early_stop_counter = 0
        torch.save(model.state_dict(), 'best_model.pth')
    else:
```

```
        early_stop_counter += 1

    if early_stop_counter >= patience:
        print("Early stopping triggered.")
        break
```

3 - Generate Results / test

In [ ]:
```
test_loader = DataLoader(
    PostDataset(test_data['text'], [0] * len(test_data), tokenizer, max_len)
    batch_size=64, num_workers = NUM_WORKERS, pin_memory=True
)



# Load the best model
model = AutoModelForSequenceClassification.from_pretrained('vinai/bertweet-b

# Wrap the model in DataParallel
model = torch.nn.DataParallel(model)

# Load the state_dict
model.load_state_dict(torch.load('best_model.pth'))

model.to(device)
model.eval()
test_predictions = []

with torch.no_grad():
    for batch in test_loader:
        # load to gpu
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)

        # Predict
        outputs = model(input_ids, attention_mask=attention_mask)
        logits = outputs.logits
        preds = torch.argmax(logits, dim=1)
        test_predictions.extend(preds.cpu().numpy())

# retrieve original labels and output
test_data['emotion'] = le.inverse_transform(test_predictions)
submission = test_data[['tweet_id', 'emotion']]
submission = submission.rename(columns={'tweet_id': 'id'})
submission.to_csv('/kaggle/working/submission.csv', index=False)

print('Submission saved successfully!')
```

4 - Observations

This model is highly computationally intensive and could benefit from more optimizations. I tried implementing it to use TPU, but there was a bug with the transformers library causing the environment to crash.

To improve the model more data could be generated, for example by using word2vec to replace words with synonyms or use LLMs to rephrase sentences. This would be an expensive task but that could prove beneficial.

```
In [ ]:
```