Natural Language Processing Assignment

Topic: Named Entity Recognition Using BERT

NAME:

SOUVAGYA GHOSH (2021AAPS1485P)

KRTEYU PILLAI (2021A7PS2522P)

Data files Used in our Program

The dataset that our group chose was “wnut-17”.

This model can be broken down into 3 sets: Training, Testing, and Validation.

The data is split with size 3394, 1009, and 1287 respectively. Our NLP Model uses BERT to pre process the data and train it. BERT is already pretrained, and consists of a library of its own with a very huge size.

If a word is not found in its dictionary it can split the word into other words which get the job done. For example, let us consider the word “pretrained”. It might not consist of this word; however, it could have “pre” and trained in its assigned vocabulary or embeddings. Hence, it splits the word into two i.e., “pre” and “##trained”, adding 2 hashes Infront of the second word to indicate that it is the continuation of the previous word. While dealing with a sentence, it also adds 2 special tokens, [CLS] and [SEP] at the beginning and the end of the sentence respectively.

However, the model takes in 34 inputs, so it requires padding. Also, we need to get rid of the special characters, as well as hashing so as to input these into the neural networks as embeddings and output the desired result.

Libraries Used

The Libraries used here:

```
from datasets import load_dataset
from transformers import AutoTokenizer
import torch
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
from torch.nn.utils.rnn import pad_sequence
from transformers import BertForTokenClassification, AdamW
from transformers import BertConfig
from transformers import DataCollatorForTokenClassification
from torch.optim.lr_scheduler import StepLR
from torch.optim import SGD
import evaluate
import numpy as np
from torch.optim.lr_scheduler import StepLR
from torch.nn.utils import clip_grad_norm_
from datasets import load_dataset
from transformers import AutoTokenizer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import random
```

Overview of the File Code

The model code consists of 3 python files, and 2 result files.

The first file, consists of data pre-processing, which takes out the data, breaks it down into tokens with sufficient padding and matched labels, and then inputs it into it's second part that is the model. The model consists of various parameters and on the basis of that, tests and outputs the result.

The second python files consists of functions used in the first one. In this there are 3 functions:

1. Tokenizer Code: This returns the respective labels of the code. It assigns -100 to special tokens such as [SEP]. It basically returns the respective label which is present in the dataset, so as to give back an useful compiled ner tags.
2. Compute Metrics: As the name suggests, it returns metrics such as Precision, Accuracy, F1 Score and Recall values, which shows which model is best.
3. Random Set: This is useful for mini batch training. It takes out a fixed number of data from training and it picks random numbers, so as to ensure that the dataset is not continued but rather a random compilation, so that when mini batch training occurs, it simply does not compute results based on a range of data.

Finally, the 3rd Python file is useful for evaluation. It takes in the model parameters from the training done and computes results from the testing dataset. It then returns the precision, accuracy, etc by using the model's weights and parameters.

Data Pre-Processing: How the data is Processed

First, the data is extracted from the file. We use the training data set obviously, and process it.

```
data = load_dataset("wnut_17")
```

We now tokenize the data using the auto tokenizer function.

```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

As said earlier, it creates special tokens. We need to use a function which assigns -100 label to these special tokens (-100 means the model ignores this, as it means garbage value), and assign the label to first word from a given token, and -100 to others.

We refer to this function from the 2nd Python file:

```
def helper(data):
    input=tokenizer(data["tokens"],truncation=True, is_split_into_words=True)

    label_final=[]
    for i,label in enumerate(data[f"ner_tags"]):
        id=input.word_ids(batch_index=i)
        prev=None
        id_append=[]
        for idx in id:
            if idx is None:
                id_append.append(-100)
            elif idx!=prev:
                id_append.append(label[idx])
            else:
                id_append.append(-100)
                prev=idx
        label_final.append(id_append)
    input["labels"]=label_final
    return input
```

The is_split_into_words command splits the unknown words into it's respective subwords. It checks if the word is valid, and assigns it a label. This final label list , that is the ner tags is sent back to the first file, which maps it to it's respective data.

```
tokenized_output = data.map(tc.helper, batched=True)
```

batched=true does it faster as it takes in multiple data values at once to process and return it.

Now we have the respective NER tags to each word. We now pad the input ids and their respective labels so that we can input it into the model. The model takes in a fixed number of inputs (34) to process, so we should pad it or cut down (which results in info loss) to a size of 34.

```
padded_input_ids = pad_sequence([torch.tensor(seq) for seq in
tokenized_output["train"]["input_ids"]], batch_first=True,
padding_value=tokenizer.pad_token_id)
padded_labels = pad_sequence([torch.tensor(seq) for seq in
tokenized_output["train"]["labels"]], batch_first=True,
padding_value=tokenizer.pad_token_id)
```

We then convert it into tensors, as the model takes only tensors as a valid data type.

```
train_dataset = TensorDataset(padded_input_ids, padded_labels)
```

This concludes our data preprocessing. We extracted the data from the dataset, updated the NER values, and padded it so that we can enter it into the model for training.

Fine Tuning the Model and Adjusting the Parameters

The model requires a few parameters to set before we can actually start the training.

We first set the label to ID and ID to Label parameters, which are used in the configuration of the model.

```
id_label = {
    0: "O",
    1: "B-corporation",
    2: "I-corporation",
    3: "B-creative-work",
    4: "I-creative-work",
    5: "B-group",
    6: "I-group",
    7: "B-location",
    8: "I-location",
    9: "B-person",
    10: "I-person",
    11: "B-product",
    12: "I-product",
}
label_id  = {
    "O": 0,
    "B-corporation": 1,
    "I-corporation": 2,
    "B-creative-work": 3,
    "I-creative-work": 4,
    "B-group": 5,
    "I-group": 6,
    "B-location": 7,
    "I-location": 8,
    "B-person": 9,
    "I-person": 10,
    "B-product": 11,
    "I-product": 12,
}
```

These are used in the config of the code.

Next comes the important parameters used for the model building.

This involves the following:
1. Learning Rate: An important parameter. The lower the better, as it determines the step size to move towards a lower loss. Having a high LR can result in bad performance. Given such high number of data sets, having a lower learning rate can have precise movement towards the lower loss function.

```
lr = 1e-5
```

2. Epoch Size: The number of iterations this code is ran through. After each iteration, the weights are retained. However, after one point, it starts to saturate, which results in time wastage of time and resources as the weights don't change much.

```
epoch_sz = 4
```

3. Batch Size: This is basically the equivalent of taking in multiple inputs at a time and adjusting weight depending on that. This helps the model converge to a better solution.

```
train_batch_size = 12
```

4. Weight Decay: This serves as a regularisation constant.

```
w_dec = 0.01
```

Other things such as Hidden Layer Size, Number of Hidden Layers are seen later when the code is ran.

Configuration of The Code

Now comes configuring the code. This is where the BERT model is used for token classification, and the id2label and label2id are the previous ids defined.

```
config = BertConfig.from_pretrained("bert-base-uncased", num_labels=13,
id2label=id_label, label2id=label_id)
```

We can adjust the code to modify the architecture, such as modifying the number of hidden layers, size of hidden layers, etc. This serves as a basis of modifying and seeing how hidden layers affect the code.

Structure of The Model

This involves the structure of the model. It begins off by writing a few parameters of the model, which is the configuration of the model, the optimizer, the Loss function we use, and at last the optimizer. Several other parameters can be defined which help improve the performance of the code, such as scheduler which is introduced at the end of the code.

```
model = BertForTokenClassification(config)
optimizer = AdamW(model.parameters(), lr=lr, weight_decay=w_dec)
loss_fn = torch.nn.CrossEntropyLoss()
```

Training Loop

The training starts now.

First, we run the code in a loop with a desired number of epochs.

```
for epoch in range(epoch_sz):
```

We define the loss, precision, f1, recall, etc and all are set to 0. These metrics are averaged over each epoch, to represent the respective data per epoch. Other things such as norm is also written, which is later used in the code for gradient clipping.

```
loss_avg=0
precision=0
f1=0
recall=0
accuracy=0
num=0
max_norm=1.0
```

The model.train() starts the training.

```
model.train()
```

Inside this, we run another loop, which is passing each input into the code. It takes in input based off on the size of batches, which this code has used 12 for each, approximating upto 283 iterations.

```
for batch in DataLoader(train_dataset, batch_size=train_batch_size,
shuffle=True):
        #check target size matches the desired size, else it will throw an
error
        inputs, targets = batch
        assert targets.max() < num_labels
        #forward pass
        outputs = model(input_ids=inputs, labels=targets)


        optimizer.zero_grad()



        loss = loss_fn(outputs.logits.view(-1,13), targets.view(-1))
        loss_avg+=loss
        loss_avg/=num_labels

        #preparing parameters for computing metrics
        pred=outputs.logits.argmax(dim=-1).view(-1)
        label=targets.view(-1)


        #computing loss, precision, f1 score, accuracy
        num+=1

        precision+=tc.compute_metrics(preds=pred,labels=label)["precision"]
        accuracy+=tc.compute_metrics(preds=pred,labels=label)["accuracy"]
        f1+=tc.compute_metrics(preds=pred,labels=label)["f1"]
        recall+=tc.compute_metrics(preds=pred,labels=label)["recall"]


        print(f"itr: {num}")
        print(f"p:{precision/num}, a:{accuracy/num}")
```

```
        #backward pass
        loss.backward()
        #gradient clipping
        clip_grad_norm_(model.parameters(),max_norm)

        optimizer.step()
```

Lastly, the parameters are computed by averaging them up.

This data is written into a text file, which is used for judging how good the model variant is.

```
      loss_avg=loss_avg/num
    L=f"For Epoch {epoch+1}, Loss Avg = {loss_avg}, Precision={precision/num},
Accuracy={accuracy/num}, f1 score={f1/num}, recall={recall/num}, learning
rate={lr}"
    print(L)
    file1=open("records.txt", "a")
    file1.write(f"For Epoch {epoch+1}, Loss Avg = {loss_avg},
Precision={precision/num}, Accuracy={accuracy/num}, f1 score={f1/num},
recall={recall/num}, learning rate={lr} , Hidden Layer Size={H}, Hidden
Layers={H_L}\n")
    file1.close()
```

MODEL VARIANTS

NUMBER OF EPOCHS

We first start off by adjusting the number of epochs.

We computed the metrics at each epoch. Number of epochs was set to 4. The results that were compiled are:

Epoch 1:

For Epoch 1, Loss Avg = 0.1431722193956375, Precision=0.9134521678515456, Accuracy=0.953212406753043, f1 score=0.9321702020079806, recall=0.953212406753043, learning rate=1e-05

Epoch 2:

For Epoch 2, Loss Avg = 0.12593689560890198, Precision=0.9134540463290137, Accuracy=0.9557000392618765, f1 score=0.9340763382690159, recall=0.9557000392618765, learning rate=1e-05

Epoch 3:

For Epoch 3, Loss Avg = 0.10725441575050354, Precision=0.9134230285739205, Accuracy=0.9556819787985871, f1 score=0.9340506139693713, recall=0.9556819787985871, learning rate=1e-05

Epoch 4:

For Epoch 4, Loss Avg = 0.10485193133354187, Precision=0.9134545410286601, Accuracy=0.9556961130742053, f1 score=0.9340727091238686, recall=0.9556961130742053, learning rate=1e-05

From this, we infer that apart from Loss, the rest of the metrics did not have a significant change. However, Loss in Epoch 1 fell from 0.14 to 0.12 in Epoch 2 to 0.107 in Epoch 3. It is true that using multiple epochs makes the model work better, as the loss fell down in each Epoch. However, by Epoch 4, the loss barely got lesser, which suggests that even though epochs increases the performance of a model, after a certain time it starts to saturate.

That is, using 100 Epochs or 1000 Epochs wouldn't have much of a difference in their performance.

VARYING THE LEARNING RATE

We tested the model for learning rates through 1e-1 to 1e-5.

Results:

LR=0.1

For Epoch 1, Loss Avg = 2.1854586601257324, Precision=0.9070523993084311, Accuracy=0.9355704750687089, f1 score=0.9174442257496244, recall=0.9355704750687089, learning rate=0.1

For Epoch 2, Loss Avg = 0.1983657330274582, Precision=0.9134330598761783, Accuracy=0.9556521397722809, f1 score=0.9340412651676676, recall=0.9556521397722809, learning rate=0.1

LR=0.01

For Epoch 1, Loss Avg = 0.13487575948238373, Precision=0.9134449462984767, Accuracy=0.9556937573616016, f1 score=0.9340678739722361, recall=0.9556937573616016, learning rate=0.01

For Epoch 2, Loss Avg = 0.13430963456630707, Precision=0.913430101470138, Accuracy=0.9556906164114641, f1 score=0.934060829488631, recall=0.9556906164114641, learning rate=0.01

LR=1E-3

For Epoch 1, Loss Avg = 0.14510174095630646, Precision=0.9127128576538837, Accuracy=0.9522983902630545, f1 score=0.9307470466434756, recall=0.9522983902630545

For Epoch 2, Loss Avg = 0.13505974411964417, Precision=0.9134515028573913, Accuracy=0.9556968983117395, f1 score=0.934072705483033, recall=0.9556968983117395

LR=1E-4

For Epoch 1, Loss Avg = 0.10314669460058212, Precision=0.9187620442505081, Accuracy=0.9536207302709071, f1 score=0.934097468321732, recall=0.9536207302709071, learning rate=0.0001

For Epoch 2, Loss Avg = 0.044442787766456604, Precision=0.9330828137676516, Accuracy=0.9624365920691001, f1 score=0.9468439278565317, recall=0.9624365920691001, learning rate=0.0001

LR=1E-5

(This is the standard LR for comparing it to other variants of the model)

For Epoch 1, Loss Avg = 0.1593087911605835, Precision=0.9139067628783745, Accuracy=0.9464774244208877, f1 score=0.9270604144157173, recall=0.9464774244208877

For Epoch 2, Loss Avg = 0.06817394495010376, Precision=0.9213437021522625, Accuracy=0.9581067923046719, f1 score=0.9388947355369877, recall=0.9581067923046719

This suggests that the metric values increased as Learning rate was divided by 10. As we go want to reach the minimum value of a loss function, and in a very large dataset, having a low learning rate is better. This is because stepping larger may cause us to miss the lowest value, which stepping smaller lets us reach the lowest value with more precision.

Its like, lets say man 1 and man 2 have to reach a Place 1, which is 11 meters further from them. Man 1 can move 20 meters at once, while man 2 can move 2 meters at once. They can move only 10 steps. Obviously man 2 is closer to the destination. Similar case with Learning rate, given that the data set is large enough.

The loss also decreases, as expected. However, we can also see that LR= 1E-4 is the perfect learning rate in this tool. Smaller LR fail to make any significant changes.

HIDDEN LAYER SIZE H

The hidden layer size H allows the model to have a larger capacity. However, the time and computation resources are increased a lot, which results in more computational time.

The results are:

Standard H size=768

H=384

For Epoch 1, Loss Avg = 0.16250810027122498, Precision=0.9134795275225912, Accuracy=0.950583431488025, f1 score=0.9299384819462347, recall=0.950583431488025, learning rate=1e-05 , Hidden Layer Size=384

For Epoch 2, Loss Avg = 0.11887376010417938, Precision=0.9134549830301427, Accuracy=0.9556984687868086, f1 score=0.9340750962096375, recall=0.9556984687868086, learning rate=1e-05 , Hidden Layer Size=384

H=1152

For Epoch 1, Loss Avg = 0.1453377902507782, Precision=0.9101546074006792, Accuracy=0.9522897526501762, f1 score=0.9307206463416081, recall=0.9522897526501762, learning rate=1e-05 , Hidden Layer Size=1152

For Epoch 2, Loss Avg = 0.11349555850028992, Precision=0.9134385026392692, Accuracy=0.9556945425991364, f1 score=0.9340668954733705, recall=0.9556945425991364, learning rate=1e-05 , Hidden Layer Size=1152

As observed, the loss average between H = 384 and H=1152 becomes much more clear in the first Epoch. However, it is kind of insignificant in Epoch 2, with only a difference of roughly 0.05. This is because after multiple epochs, it saturates, as weight does not get updated much: it is near the optimum value.

NUMBER OF HIDDEN LAYERS

Results:

Hidden Layer= 18

For Epoch 1, Loss Avg = 0.14102865755558014, Precision=0.9135732179906634, Accuracy=0.952307027875933, f1 score=0.9307611374581647, recall=0.952307027875933, learning rate=1e-05
For Epoch 2, Loss Avg = 0.10198768764734268, Precision=0.9134473192863053, Accuracy=0.9556929721240677, f1 score=0.9340677366125509, recall=0.9556929721240677, learning rate=1e-05

Hidden Layer=6

For Epoch 1, Loss Avg = 0.16518192410469055, Precision=0.9136545786638968, Accuracy=0.9519787985865725, f1 score=0.9310870129131369, recall=0.9519787985865725, learning rate=1e-05
For Epoch 2, Loss Avg = 0.12850163209438324, Precision=0.9134586526630207, Accuracy=0.9556929721240676, f1 score=0.9340726684606778, recall=0.9556929721240676, learning rate=1e-05

The loss average difference between them is significant, attributing to the number of layers.

However, the computation time significantly increased during the computation of hidden layers = 18, which makes sense, considering the layers and processing increased a lot.

Code:

```
config.num_hidden_layers=6
config.hidden_size=1152
```

Another thing to note is that hidden size should be a multiple of a specific number designated by the NLP code.

REGULARIZATION CONSTANT

You achieve this by altering the weight decay. The point of regularization constant is to penalise the weights. Weight decay is the perfect tool for this. As a result, we use this to compare if Regularization model makes any difference anyways.

For weight decay=0.0

For Epoch 1, Loss Avg = 0.14095835387706757, Precision=0.913372831707227, Accuracy=0.9524986258343153, f1 score=0.931101177434131, recall=0.9524986258343153, learning rate=1e-05

For Epoch 2, Loss Avg = 0.11040251702070236, Precision=0.9134838775844656, Accuracy=0.9557078916372196, f1 score=0.9340911181590872, recall=0.9557078916372196, learning rate=1e-05

For weight decay=0.1:

For Epoch 1, Loss Avg = 0.1593087911605835, Precision=0.9139067628783745, Accuracy=0.9464774244208877, f1 score=0.9270604144157173, recall=0.9464774244208877

For Epoch 2, Loss Avg = 0.06817394495010376, Precision=0.9213437021522625, Accuracy=0.9581067923046719, f1 score=0.9388947355369877, recall=0.9581067923046719

As seen, with a regularization constant, the loss is much more less. Hence, weight decays are useful, since penalising weights help adjust the weights because of which the optimum weights could be found.

LOSS FUNCTION

2 loss functions are used: Cross Entropy Loss and L1 Loss. Cross Entropy Loss is better suited for such a model, since it finds a good balance between the metrics and the loss function. When metrics are high, it suggests that the according to the loss, the weights are being adjusted accordingly. L1 computes a relatively high loss, however the accuracy and other results are down the drain.

Cross Entropy Loss:

For Epoch 1, Loss Avg = 0.1593087911605835, Precision=0.9139067628783745, Accuracy=0.9464774244208877, f1 score=0.9270604144157173, recall=0.9464774244208877

For Epoch 2, Loss Avg = 0.06817394495010376, Precision=0.9213437021522625, Accuracy=0.9581067923046719, f1 score=0.9388947355369877, recall=0.9581067923046719

For Epoch 1, Loss Avg = 0.16225184127, Precision=0.9207294562828121, Accuracy=0.3091762858264626, f1 score=0.4610849334289779, recall=0.3091762858264626, learning rate=1e-05

For Epoch 2, Loss Avg = 0.15824065543, Precision=0.919429781459076, Accuracy=0.30864153906556757, f1 score=0.4603835366951301, recall=0.30864153906556757, learning rate=1e-05

For Epoch 3, Loss Avg = 0.15613973047, Precision=0.920624376583852, Accuracy=0.3092194738908518, f1 score=0.46119522565962195, recall=0.3092194738908518, learning rate=1e-05

For Epoch 4, Loss Avg = 0.15663970913, Precision=0.9204747853805832, Accuracy=0.30741264232430315, f1 score=0.45925345073408225, recall=0.30741264232430315, learning rate=1e-05

L1 computes mean error, hence fails to encapsulate the whole picture. The accuracy is too low, so as the f1 and recall.

Code:

To code L1 Loss, there had to be made a few changes. The way the loss was computed was to take loss of the input for all the classes and summed up. The output had to be flattened using a for loop.

```
      for i in range(num_labels):
            new_output=outputs.logits[:,:,i]

            loss = loss_fn(new_output.view(-1), targets.view(-1))
            loss_avg+=loss
```

OPTIMIZER

The optimizer this code used was AdamW. We compared it with the SGD (Stochastic Gradient Descent).

The results are as shown below:

SGD:

For Epoch 1, Loss Avg = 1.0441408157348633, Precision=0.9114289787740131, Accuracy=0.8261452689438558, f1 score=0.8339591730253156, recall=0.8261452689438558, learning rate=1e-05

For Epoch 2, Loss Avg = 0.26230186223983765, Precision=0.9134251801247641, Accuracy=0.9556804083235185, f1 score=0.9340497224526215, recall=0.9556804083235185, learning rate=1e-05

AdamW:

For Epoch 1, Loss Avg = 0.1593087911605835, Precision=0.9139067628783745, Accuracy=0.9464774244208877, f1 score=0.9270604144157173, recall=0.9464774244208877

For Epoch 2, Loss Avg = 0.06817394495010376, Precision=0.9213437021522625, Accuracy=0.9581067923046719, f1 score=0.9388947355369877, recall=0.9581067923046719

The loss is too high in comparison to AdamW, which could be attributed to several factors such as it's adaptive learning rates, momentum building, lacking a learning rate warmup, etc.

SINGLE TRAINING ITEM VS MINI BATCH

The code uses batches to process the output. This is beneficial from single batch passing as the model gets the context of multiple batches at once. It then averages them out, and gets a better understanding of how to adjust the weights. It also helps generalize unseen data.

It provides the model with a smoothened gradient, making the optimization stable. This is not seen in Single Training item.

Single Training Item:

For Epoch 1, Loss Avg = 7.551788712589769e-06, Precision=0.9144923328750468, Accuracy=0.9554154390100458, f1 score=0.9341119558991666, recall=0.9554154390100458, learning rate=1e-05 , Hidden Layer Size=1152, Hidden Layers=18

For Epoch 2, Loss Avg = 1.1797071238106582e-05, Precision=0.9144819223466648, Accuracy=0.9556904340994189, f1 score=0.9343583091429826, recall=0.9556904340994189, learning rate=1e-05 , Hidden Layer Size=1152, Hidden Layers=18

The loss is too much, attributing to poor performance.

FREEZING LAYERS

Code:

```
for param in model.bert.embeddings.parameters():
    param.requires_grad = False
```

This freezes the embedding layer. If only the embedding layer is frozen, the output we get is:

For Epoch 1, Loss Avg = 0.1417665034532547, Precision=0.9129436802600353, Accuracy=0.9523431488025128, f1 score=0.930839118726574, recall=0.9523431488025128, learning rate=1e-05

For Epoch 2, Loss Avg = 0.1147407814860344, Precision=0.9134353232997419, Accuracy=0.9556835492736552, f1 score=0.9340554133385839, recall=0.9556835492736552, learning rate=1e-05

Freezing the embedding as well as transformer layer:

```
for param in model.bert.encoder.layer[:3].parameters():
        param.requires_grad = False
```

This freezes the transformer layer from 1 to 3. The output we get is:

For Epoch 1, Loss Avg = 0.14248597621917725, Precision=0.9136000701669138, Accuracy=0.9523188064389476, f1 score=0.9309736784061371, recall=0.9523188064389476, learning rate=1e-05

For Epoch 2, Loss Avg = 0.11613662540912628, Precision=0.9134448545129334, Accuracy=0.9556968983117392, f1 score=0.9340708607068625, recall=0.9556968983117392, learning rate=1e-05

The loss difference between freezing embedding layers and freezing both embedding and transformer layers is visible. Although minimal, it suggests that freezing leads to a greater loss. This is because for a pretrained model BERT, if layers are frozen which learn specific features of the task, it may struggle to adapt to those features. It is preferable to freeze lower layers, as they capture more generic features and enables the model to focus on specific learning the features.

USING A SMALLER BATCH

From the training data, a fixed number of randomised data is collected. This is done through the random_set() function defined in the second file. The code is:

```
def random_set(val,array,array2):

    random_indices=random.sample(range(len(array)),val)
    new_arr=[]
    new_arr1=[]

    for i in random_indices:
        new_arr.append(array[i])
        new_arr1.append(array2[i])
    return new_arr,new_arr1
```

The padded input ids and padded labels are modified as such:

```
padded_input_ids,padded_labels=tc.random_set(1024,padded_input_ids,padded_labe
ls)

padded_input_ids_tensor = torch.stack(padded_input_ids)
padded_labels_tensor = torch.stack(padded_labels)
```

This way, a fixed number of 1024 data sets are collected. The parameters are as follows:

For Epoch 1, Loss Avg = 0.17477479577064514, Precision=0.9141468988368212, Accuracy=0.9458656330749351, f1 score=0.9254317366407242, recall=0.9458656330749351, learning rate=1e-05

For Epoch 2, Loss Avg = 0.12495043873786926, Precision=0.914749354005168, Accuracy=0.9563824289405686, f1 score=0.9350818270819033, recall=0.9563824289405686, learning rate=1e-05

This suggests that when we use a smaller data set, the loss is higher since there is lesser diversity than when we use the complete data set. As a result, we get higher loss.

OPTIMIZING THE CODE FURTHER

Step Scheduler

This function takes in 2 parameters, gamma and step size. After a fixed number of step size, it multiplies the current learning rate by gamma. This is much more optimized as by adjusting the learning rate midway is much more convenient as it allows for faster convergence.

Code:

```
scheduler = StepLR(optimizer, step_size=1, gamma=0.1)
scheduler.step()
```

Gradient Clipping

This prevents the gradient from exploding, if they ever become too large. This is more of a precaution optimizing. If the gradient becomes too large, training process becomes difficult and leads to numerical instability.

The gradient is rescaled to ensure its norm stays within a fixed range.

Code:

```
clip_grad_norm_(model.parameters(),max_norm)
```

Incorporating all these results and optimizing codes, the output we get is:

For Epoch 4, Loss Avg = 0.04848666489124298, Precision=0.9299043131550803, Accuracy=0.9612972124067533, f1 score=0.9447931373699092, recall=0.9612972124067533

This is the optimum result.

EVALUATION

The evaluation code involves a few changes, otherwise it is pretty similar to the first one.

The model derived from training is saved into a file. Through that, we load the model and use it for evaluation.

Code:

How we load it:

```
model=BertForTokenClassification.from_pretrained("bert-base-uncased")
loaded_model = torch.load('weight.pth', map_location=torch.device('cpu'))
```

The data tokenization process is also the same. We start off from model.eval().

The evaluation loop is:

```
model.eval()
model.config.num_labels=13
import torch

loss_fn = torch.nn.CrossEntropyLoss()
num = 0
loss_avg = 0.0
precision = 0.0
accuracy = 0.0
f1 = 0.0
recall = 0.0

with torch.no_grad():
    for batch in test_data:
        num += 1
        inputs, targets = batch


        attention_mask = (inputs != tokenizer.pad_token_id).float()
        outputs = model(input_ids=inputs, attention_mask=attention_mask)



        pred = outputs.logits.argmax(dim=-1).view(-1)
        label = targets.view(-1)

        precision += tc.compute_metrics(preds=pred, labels=label)["precision"]
        accuracy += tc.compute_metrics(preds=pred, labels=label)["accuracy"]
        f1 += tc.compute_metrics(preds=pred, labels=label)["f1"]
        recall += tc.compute_metrics(preds=pred, labels=label)["recall"]
```

```
        print(f"Iteration: {num}")
        print(f"Precision: {precision / num}, Accuracy: {accuracy / num}")

loss_avg /= num
precision /= num
accuracy /= num
f1 /= num
recall /= num

print(f"The evaluation metrics are: Precision = {precision}, Accuracy =
{accuracy}, F1 score = {f1}, and Recall = {recall}")
```

RESULTS:

We used the weights from training, and following are the results we got:

The evaluation metrics are: Precision = 0.9620533404290581, Accuracy = 0.9179513282916425, F1 score = 0.9392930830610394, and Recall = 0.9179513282916425

This suggests that the metrics derived from the testing of the data is high, which affirms that the weights obtained via training are highly accurate.

CONCLUSION

There are a lot of things which could be implemented further into this code. Nevertheless, this code managed to implement many features such as StepScheduler, Grad Clipping, adjusting/freezing layers, etc

On the basis of the results obtained, the most optimum result was chosen, which proved to be accurate since Metrics obtained were very high in terms of percentage.

While training, this code tweaked a few parameters and compared the results to other results, which helped deduce implementing which code would be suitable for the dataset.

END