

```
%%capture
!pip install datasets
!pip install evaluate
```

```
import pandas as pd, fastai
```

✓ Gradient Accumulation and Ensembling Deep Learning Models

✓ Introduction

In this notebook, I will be making a submission to the following Kaggle competition: [Natural Language Processing with Disaster Tweets](#).

The goal is to use **natural language processing** to predict if a tweet is talking about a real natural disaster or not. If `target = 1`, the tweet is talking about a real disaster. If `target = 0` the tweet is **not** talking about a real disaster.

In line with FastAI's lesson 7, I will be using this competition as an opportunity to learn new machine learning skills, including:

- **gradient accumulation**, and,
- **ensembling** a number of larger pretrained NLP models.

✓ Data Processing

✓ Collecting Data


Let's start by having a look at our training data.

```
df = pd.read_csv('train.csv')
df
```



	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1
...
7608	10869	NaN	NaN	Two giant cranes holding a bridge collapse int...	1
7609	10870	NaN	NaN	@aria_ahrany @TheTawniest The out of control w...	1
7610	10871	NaN	NaN	M1.94 [01:04 UTC]?5km S of Volcano Hawaii. htt...	1

```
df_test = pd.read_csv('test.csv')
df_test
```



	id	keyword	location	text
0	0	NaN	NaN	Just happened a terrible car crash
1	2	NaN	NaN	Heard about #earthquake is different cities, s...
2	3	NaN	NaN	there is a forest fire at spot pond, geese are...
3	9	NaN	NaN	Apocalypse lighting. #Spokane #wildfires
4	11	NaN	NaN	Typhoon Soudelor kills 28 in China and Taiwan
...
3258	10861	NaN	NaN	EARTHQUAKE SAFETY LOS ANGELES ⚠️ SAFETY FASTE...
3259	10865	NaN	NaN	Storm in RI worse than last hurricane. My city...
3260	10868	NaN	NaN	Green Line derailment in Chicago http://t.co/U...
3261	10874	NaN	NaN	MEG issues Hazardous Weather Outlook (HWO) htt...
3262	10875	NaN	NaN	#CityofCalgary has activated its Municipal Eme...

3263 rows × 4 columns

My first note is that there appears to be a number of empty data points. We can have a deeper dive into this.

At this point, let's define our variables:

- `text` is the text of a tweet.
- `keyword` is a keyword from that tweet (although this may be blank!).
- `location` is the location the tweet was sent from (may also be blank).

```
df.isnull().sum()
```



	0
<hr/>	
id	0
keyword	61
location	2533
text	0
target	0

dtype: int64

```
df_test.isnull().sum()
```



	0
<hr/>	
id	0
keyword	26
location	1105
text	0

dtype: int64

We have lots of empty data points!

Since the fact `location` or `keyword` is empty could be useful for the model, we will replace `NaN` values with the string `'empty'`.

```
df.fillna("empty", inplace=True)
```

Because we will be using `Transformers` to create our model, we need to relabel our `target` column to `labels`.

```
df.rename(columns={'target':'labels'}, inplace=True)
```

✓ Feature Engineering

At this point, we want to combine all our features into a single input string that we can tokenize and numericalize.

```
df["text"] = "KEYWORD:" + df.keyword + "LOCATION:" + df.location + "TEXT:" + df
df.drop(['id', 'keyword', 'location'], axis=1, inplace=True)
df.head()
```



	text	labels
0	KEYWORD:emptyLOCATION:emptyTEXT:Our Deeds are ...	1
1	KEYWORD:emptyLOCATION:emptyTEXT:Forest fire ne...	1
2	KEYWORD:emptyLOCATION:emptyTEXT:All residents ...	1
3	KEYWORD:emptyLOCATION:emptyTEXT:13,000 people ...	1
4	KEYWORD:emptyLOCATION:emptyTEXT:Just got sent ...	1

✓ Validation Set

We are going to be using a random subset of data as our validation set, which can separate using the Hugging Face `datasets` library.

```
from datasets import Dataset

dataset = Dataset.from_pandas(df)

data = dataset.train_test_split(test_size=0.2)
train = data['train']
val = data['test']
```

✓ Memory and Gradient Accumulation

We are going to be **ensembling** 4 different pretrained NLP models from Transformers. I've defined them below.

```
models = ["distilbert-base-uncased", "bert-base-uncased", "roberta-base", "xlm-ro"]
```

But first, I'm going to introduce the idea of **gradient accumulation**. Then, I'm going to use

gradient accumulation to enable the ensembling of these 5 models, some of which are quite large. This will allow me to run them on Colab's free (and somewhat limited) GPU!

This [Kaggle notebook](#) goes into incredible detail about what gradient accumulation is. If I was using fastai, I could simulate the steps, but it is more challenging (and not really necessary) in Transformers. Instead I will describe it, since I've used gradient accumulation below.

Essentially, the variable `accum` will divide the batch size by this value. Rather than updating the model's weights after every batch, we will keep **accumulating** the gradients (specifically `accum` times)!

This explains why in PyTorch, when we create the loss function manually we need this line:

```
coeffs.grad.zero_()
```

Without this line, the gradients will automatically accumulate. So, when we define the `accum` variable, we will only call `zero_()` when we have completed a **full** batch, like so:

```
count = 0

for x,y in dl:
    count += len(x)
    calc_loss(coeffs, x, y).backward()

    if count >= batch_size:
        coeffs.data.sub_(coeffs.grad * lr)
        coeffs.grad.zero_()
        count=0
```

Why is this useful? Well, mathematically, the training loop is nearly identical to when `accum=1`. However, the amount of memory used by the GPU will be much smaller as it is not working out an enormous number of gradients at the same time.

This is fantastic because more expensive GPUs generally have more memory, **but not necessarily much more performance**. This is a really cost-effective way of simulating the performance of larger GPUs without actually needing their memory.

Why don't we just use a smaller batch size? Well, larger batches mean the model updates the weights less frequently. This means that the average gradient is less susceptible to noise, as it is calculated from a larger number of parameters. This can reduce the chance of **overfitting**

and improve the model's ability to generalise.

We will be using **gradient accumulation** with some of our models when we ensemble them, as these larger pretrained models require significant memory, which we don't have with Colab's free GPU.

✓ Ensembling and Weighted Models

Now, we need to run all 4 models and ensemble their predictions! I went into more detail in [this notebook](#) about how to do NLP.

✓ Tokenization and Numericalization

First, we need to tokenize our training and validation sets. This will turn our string of words into a series of numbers representing words, subwords, or characters. It is important to note that we need a separate tokenized input for each model, as each pretrained model will tokenize in its own way. If we don't tokenize in the same way as the pretrained model, that's obviously not going to work!

- `tokenize_and_add_labels(batch, tokenizer)` extracts the text data with `batch['text']` and tokenizes it using the `tokenizer()` function. This is returned with corresponding labels from `batch["labels"]`.
 - `**` unpacks the key-value pairs from the dictionary, and merges it with the dictionary we will be returning.
 - `padding` ensures all tokenized inputs have the same length, by padding shorter sentences with `0`.
 - `truncation` cuts off extra tokens if a sentence is longer than the maximum model size (i.e., BERT has maximum of 512 tokens).
- `AutoTokenizer.from_pretrained` finds the tokenizer used by the model.
- `.map()` applies `tokenize_and_add_labels()` to each batch of data in `train` and `val`, as we have the parameter `batched` set to `True`.
- `train_tokenized` and `val_tokenized` are dictionaries containing the tokenized inputs for each of the 5 models.

```
%%capture
from transformers import TrainingArguments, Trainer
from transformers import AutoTokenizer

def tokenize_and_add_labels(batch, tokenizer):
    return {**tokenizer(batch['text'], padding='max_length', truncation=True),

train_tokenized = {}
val_tokenized = {}

for name in models:
    tokenizer = AutoTokenizer.from_pretrained(name)

    train_tokenized[name] = train.map(lambda x: tokenize_and_add_labels(x, tokenizer))
    val_tokenized[name] = val.map(lambda x: tokenize_and_add_labels(x, tokenizer))
```

✓ Training the Models

Now, we can train all 5 models, saving them in a list called `trained`.

We will define a function to compute the accuracy of our predictions, so we can evaluate the performance of the model with a metric (rather than just with loss). Hugging Face define an accuracy metric in their `evaluate` library.

Then, we define our `TrainingArguments` which are the hyperparameters we feed into the model. I played around with these to optimise the models performance.

The training is done in a `for` loop and after each iteration we use `gc.collect()` to run Python's garbage collector to free up unused objects in RAM and `torch.cuda.empty_cache()` to clear GPU memory, allowing us to train the next model without running out of GPU storage.

- `AutoModelForSequenceClassification.from_pretrained()` is used to load the pretrained model. `num_labels=2` tells the model we are doing binary classification.
- `trainer` sets up Hugging Face's Trainer API for training, validation and saving models.
 - `model` is what we are training.
 - `training_args` are predefined above.
 - `train_dataset` is the preprocessed training dataset.
 - `eval_dataset` is the preprocessed validation dataset.
 - `compute_metrics` is what we will be printing after each epoch.
- we can train the trainer, then save it to our `trained` list.

```
from transformers import AutoModelForSequenceClassification
import evaluate # Hugging Face's metric library
import numpy as np
import torch, gc

trained = []
accuracy_metric = evaluate.load("accuracy") # Load accuracy metric

def accuracy(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1) # Convert logits to class predict
    return accuracy_metric.compute(predictions=predictions, references=labels)

training_args = TrainingArguments(
    'outputs', # directory where training results, model checkpoints and logs s
```

```

eval_strategy="epoch", # how often the model is evaluated with the metric
warmup_ratio=0.1, # warm-up is a technique where the model starts with a low
lr_scheduler_type='cosine', # defines how the learning rate changes over time
gradient_accumulation_steps=8, # gradient accumulation - discussed above
learning_rate=1e-5, # learning rate
fp16=True, # mixed precision training, 16-bit floating point. Speeds up training
per_device_train_batch_size=32, # number of samples processed per GPU per step
per_device_eval_batch_size=128, # batch size for validation and updating parameters
num_train_epochs=4, # epochs
weight_decay=0.1, # weight decay
report_to='none' # reduces logging
)

for name in models:
    model = AutoModelForSequenceClassification.from_pretrained(name, num_labels=2)

    trainer = Trainer(
        model,
        training_args,
        train_dataset=train_tokenized[name],
        eval_dataset=val_tokenized[name],
        compute_metrics=accuracy
    )

    trainer.train()
    trained.append(trainer)

gc.collect()
torch.cuda.empty_cache()

```



Downloading builder script: 100%

4.20k/4.20k [00:00<00:00, 310kB/s]

model.safetensors: 100%

268M/268M [00:01<00:00, 229MB/s]

[92/92 05:00, Epoch 3/4]

Epoch	Training Loss	Validation Loss	Accuracy
0	No log	0.578809	0.790545
1	No log	0.442238	0.814183
2	No log	0.412558	0.820749
3	No log	0.406954	0.829284

model.safetensors: 100%

440M/440M [00:02<00:00, 242MB/s]

[92/92 09:42, Epoch 3/4]

Epoch	Training Loss	Validation Loss	Accuracy
0	No log	0.543516	0.768877

0	No log	0.592514	0.764938
1	No log	0.452935	0.808930
2	No log	0.430401	0.817466
3	No log	0.422716	0.820749

model.safetensors: 100%

499M/499M [00:03<00:00, 177MB/s]

[54/92 05:17 < 03:52, 0.16 it/s, Epoch 2.29/4]

Epoch	Training Loss	Validation Loss	Accuracy
0	No log	0.592514	0.764938
1	No log	0.443814	0.799737

[92/92 10:08, Epoch 3/4]

Epoch	Training Loss	Validation Loss	Accuracy
0	No log	0.592514	0.764938
1	No log	0.443814	0.799737
2	No log	0.392279	0.835850
3	No log	0.389122	0.837820

model.safetensors: 100%

1.12G/1.12G [00:10<00:00, 205MB/s]

[92/92 10:43, Epoch 3/4]

Epoch	Training Loss	Validation Loss	Accuracy
0	No log	0.603417	0.596192
1	No log	0.460222	0.810243
2	No log	0.453043	0.806960
3	No log	0.427917	0.826001

We have trained 4 models that don't appear to be overfitting (since the loss on the validation set consistently reduces) and are performing well! Since it took so long to train all the models, we don't want to accidentally lose them, so we will use `fastai`'s `save_pickle` function to keep them safe.

```
from fastai.data.external import save_pickle

save_pickle('models.pkl', trained)
```

✓ Submitting Predictions

Now we will make predictions on the test dataset to upload to Kaggle. We need to perform the same preprocessing on the test dataframe that we did on our training dataframe to make the predictions.

We will tokenize the test data for each model and use the `Dataset` from Hugging Face, which we need when using their API to make predictions.

NOTE: we use `map` so we can specify the `batched` parameter, which makes tokenization much faster.

Since we don't need labels with the test dataset, we don't need any dictionary unpacking.

```
df_test.rename(columns={'target':'labels'}, inplace=True)
df_test.fillna("empty", inplace=True)
df_test["text"] = "KEYWORD:" + df_test.keyword + "LOCATION:" + df_test.location
df_test.drop(['id', 'keyword', 'location'], axis=1, inplace=True)
```

```
test = Dataset.from_pandas(df_test)
```

```
test_tokenized = {}
```

```
for name in models:
    tokenizer = AutoTokenizer.from_pretrained(name)

    test_tokenized[name] = test.map(
        lambda x: tokenizer(x['text'], padding='max_length', truncation=True),
        batched=True
    )
```



Map: 100%

3263/3263 [00:01<00:00, 2288.20 examples/s]

Map: 100%

3263/3263 [00:01<00:00, 2453.93 examples/s]

Man: 100%

3263/3263 [00:01<00:00, 1649.07 examples/s]

Now, we will make our predictions, saving them in the list `preds`. The `predict()` function from Hugging Face is used on each trainer, and the predictions are casted to a float and saved a tensor (necessary when averaging our predictions later).

```
preds = []
```

```
for name, trainer in zip(models, trained):
    input = test_tokenized[name]
    pred_array = trainer.predict(input).predictions.astype(float)
    preds.append(torch.tensor(pred_array))
```



Since we have all our predictions, we will convert `preds` into a tensor by stacking each models predictions, then taking the mean along the 0th dimension (i.e., across the four models).

```
avg_preds = torch.stack(preds).mean(0)
avg_preds.shape
```

`avg_preds` contains 3,263 rows containing **logits** - raw model scores. The 0th column represents class 0 and the 1st column represents class 1.

`torch.argmax()` finds the index of the maximum value across the axis 1. This will be our predicted class and is saved in `predictions`.

```
import torch.nn.functional as F

predictions = torch.argmax(avg_preds, dim=1)

predictions
```

Now, we can combine our predictions with their `id` values into a dataframe and download this as a csv file.

```
sub_df = pd.read_csv('test.csv')
sub_df['target'] = predictions

sub_df[['id', 'target']].to_csv("submission.csv", index=False)
```

When we submitted to Kaggle, our final result was a MAPE of 0.82899, which put us in the top 25% of submissions!

✓ Conclusion

In this notebook I was able to learn in more detail about some critical machine learning concepts, including how to **ensemble** a number of larger pretrained models and how to maximise our GPU potential using **gradient accumulation**. Thank you for reading my learning journey!