NLP Assignment 2

Question Answer with SQUAD

Rajbir Bhattacharjee

R00195734

Table of Contents

[Abstract 2](#_Toc92063098)

[Running the baseline 2](#_Toc92063099)

[Memory exhaustion due to loading large pickled data structures 2](#_Toc92063100)

[Implementation of a disk-backed low-resource array 3](#_Toc92063101)

[Running the baseline 6](#_Toc92063102)

[Data Augmentation 7](#_Toc92063103)

[Masking answers within contexts 7](#_Toc92063104)

[Backtranslation Implementation 9](#_Toc92063105)

[Memory leak in Marian-MT model. 11](#_Toc92063106)

[Implementation of random-swap and random-deletion 12](#_Toc92063107)

[Results with Data Augmentation 12](#_Toc92063108)

[Slanted Triangle Learning Rate 13](#_Toc92063109)

[Results with Slanted Triangle Learning Rate 13](#_Toc92063110)

[Gradual Unfreezing 14](#_Toc92063111)

[Results with Gradual Unfreezing 16](#_Toc92063112)

[Comparison of the three approaches 17](#_Toc92063113)

[Future work 19](#_Toc92063114)

# Abstract

We attempt the task of answering questions based on the SQUAD dataset. For this model we use pre-trained DistillBert models from Huggingface. We use a provided codebase to first run a baseline after pre-training. In with the provided code-base without any change, we can see a best F1 metric of 70.67 and EM metric of 54.58. We then augment the data to double the number of training examples. The data augmentation performs slightly worse than the baseline model. We then try slanted-triangle learning rates, but the model fails to converge with this. The last experiment we try is to use gradual unfreezing, and that improves the result over the baseline achieving an F1 of 71.54 and an EM of 55.32.

During the course of this work, we faced significant hurdles in running the task. Firstly the RAM requirements of the base code provided were very high, and it required upto 60 GB of memory (RAM + SWAP). Since we wanted to run this on Google colab, we had to do significant restructuring to allow it to run. Our contribution includes development of a disk-backed array structure that can scale almost infinitely and uses much of the same interface as traditional python arrays, so they can be used with minimal code modification. Our implementation is performant especially in sequential indexing. Using this disk-based array structure, we were able to run with just 6 GB of RAM. This code has been open-sourced.

We also found out some memory leaks in the Huggingface transformers model/library which would lead to memory exhaustion when doing backtranslation during the data-augmentation phase. While we did not have the time to completely investigate this and find out the root cause, we were able to find a work-around to this. This will be discussed later.

Lastly, we found that the Huggingface transformers model/library crashes the python interpreter on certain inputs. We had to skip the training examples which lead to this crash, but finding these out was an iterative and time taking process.

# Running the baseline

We wanted to run the baseline on Google colab, but we were immediately faced by a memory issue. We used Google Colab Pro, and Colab Pro high-RAM instances give about 25GB of RAM. However, these are not backed up by any swap space, so as soon as any processes takes up 25GB of RAM, the process is killed as there is no overflow room in the swap.

## Memory exhaustion due to loading large pickled data structures

In the original code, the following code tries to load the saved encodings:

Text

Description automatically generated

Although the saved encodings themselves are about 250 megabytes in size, the process of loading them requires more than 25GB of memory and the call to pickle.load leads to memory exhaustion.

## Implementation of a disk-backed low-resource array

As a result, we decided to not keep all training instances in memory. However, we did not want to rewrite all the code, so we wanted to maintain array semantics as much as possible. We implemented a class called ChunkDataWriter. This class would keep n-elements in memory at any time. Whenever it reached the boundary of n-elements, it would create a separate pickle-file and write those n memory elements into the disk. It is complemented by a ChunkDataReader that can be used to read the files.

A screenshot of a computer

Description automatically generated with medium confidence

The class supports appending data to the end of the chunk.

Text, chat or text message

Description automatically generated

The write\_chunk() and write\_conf() functions just store the current chunk on to the disk.

The ChunkDataReader class can be used to then read these disk-based arrays.

Text

Description automatically generated

Reading the disk-based array is implemented in the \_\_getitem\_\_ function in ChunkDataReader:  
  
Text

Description automatically generated

The load\_slice method simply loads the appropriate file:

Text, chat or text message

Description automatically generated

Once this was done, the original code could be modified to use this disk-based array instead of regular pickled encodings.

The prepare-train-data function was modified in the following way to start using these chunked data instead of using pickled files.

Text

Description automatically generated

Text

Description automatically generated

In line 146, we append to the ChunkDataWriter disk-backed array instead of a regular python array.

Once this was done, we found that we were able to tokenize the entire dataset, save it and load it without any memory problems.

Since these followed array semantics for indexing, the DataLoader didn’t have to be modified.

Furthermore, the original code to tokenize the data passed all the data to the model at one go, as below:

Graphical user interface, text

Description automatically generated

However, this means that the model needs to store all the data in memory to be returned all at once. We had to break this into a loop as follows:

Text

Description automatically generated

Once all this was done, we could finally tokenize all the data in Colab, and run training on them.

Tokenizing was very time consuming, and it took almost one day to tokenize.

## Running the baseline

Once the data was tokenized and saved, the baseline could be run. We ran the baseline for 8 epochs.

[12.21.21 10:42:55] Epoch: 0

[12.21.21 10:47:11] Eval F1: 07.68, EM: 00.05

[12.21.21 11:17:40] Eval F1: 63.55, EM: 47.31

[12.21.21 11:48:08] Eval F1: 66.63, EM: 50.57

[12.21.21 12:18:39] Eval F1: 68.67, EM: 52.48

100% 242304/242304 [1:36:29<00:00, 41.86it/s, NLL=1.04, epoch=0]

[12.21.21 12:19:24] Epoch: 1

[12.21.21 12:49:09] Eval F1: 69.21, EM: 53.00

[12.21.21 13:19:39] Eval F1: 69.51, EM: 53.75

[12.21.21 13:50:09] Eval F1: 70.56, EM: 54.41

100% 242304/242304 [1:32:15<00:00, 43.77it/s, NLL=0.686, epoch=1]

[12.21.21 13:51:40] Epoch: 2

[12.21.21 14:20:41] Eval F1: 69.77, EM: 53.81

[12.21.21 14:51:11] Eval F1: 70.21, EM: 54.16

[12.21.21 15:21:41] Eval F1: 70.66, EM: 54.58

100% 242304/242304 [1:32:17<00:00, 43.76it/s, NLL=0.491, epoch=2]

[12.21.21 15:23:57] Epoch: 3

[12.21.21 15:52:12] Eval F1: 70.23, EM: 54.42

[12.21.21 16:22:44] Eval F1: 70.22, EM: 53.96

[12.21.21 16:53:18] Eval F1: 70.37, EM: 54.66

100% 242304/242304 [1:32:22<00:00, 43.72it/s, NLL=0.354, epoch=3]

[12.21.21 16:56:19] Epoch: 4

[12.21.21 17:23:53] Eval F1: 69.78, EM: 53.65

[12.21.21 17:54:25] Eval F1: 69.90, EM: 53.60

[12.21.21 18:24:53] Eval F1: 69.82, EM: 53.51

100% 242304/242304 [1:32:19<00:00, 43.74it/s, NLL=0.321, epoch=4]

[12.21.21 18:28:39] Epoch: 5

[12.21.21 18:55:21] Eval F1: 69.63, EM: 53.08

[12.21.21 19:25:48] Eval F1: 69.19, EM: 52.96

[12.21.21 19:56:16] Eval F1: 69.53, EM: 53.51

100% 242304/242304 [1:32:08<00:00, 43.83it/s, NLL=0.35, epoch=5]

[12.21.21 20:00:47] Epoch: 6

[12.21.21 20:26:45] Eval F1: 69.40, EM: 52.89

[12.21.21 20:57:13] Eval F1: 69.25, EM: 53.11

[12.21.21 21:27:45] Eval F1: 69.23, EM: 52.95

100% 242304/242304 [1:32:14<00:00, 43.78it/s, NLL=0.269, epoch=6]

[12.21.21 21:33:02] Epoch: 7

[12.21.21 21:58:17] Eval F1: 69.18, EM: 52.38

[12.21.21 22:28:51] Eval F1: 68.90, EM: 52.54

[12.21.21 22:59:21] Eval F1: 68.82, EM: 52.32

100% 242304/242304 [1:32:20<00:00, 43.73it/s, NLL=0.169, epoch=7]

[12.21.21 23:09:40] Final eval results after epoch: F1: 69.22, EM: 52.60

[12.21.21 23:09:40] After epoch, best scores so far: OrderedDict([('F1', 70.66428123994659), ('EM', 54.5803461510609)])

We can see that the best F1-score was 70.66 and the best EM score was 54.66.

# Data Augmentation

Based on several resources, we decided to augment data. To augment data, we performed the following steps:

1. Back-translation via French
2. Random swap of two words
3. Random deletion of one word

The above were applied to all questions. Backtranslation was only applied to 10% of the contexts due to time constraints, otherwise, we could have spent several days backtranslating all contexts, at the speed at which we were able to translate these items. Random deletion and random swap was applied to all contexts.

## Masking answers within contexts

We didn’t want the backtranslation process to split up our answers as that would have incurred additional processing. We therefore masked the answers in the contexts by a series of 1’s.

Text

Description automatically generated

The masked contexts looked like this:

Text

Description automatically generated

These masked texts were back translated via French.

***However, we further found that some of these masks itself were changed by the translation models. The still contained a sequence of 1’s, but the number of 1’s in them were different.***

The following snip shows the data going through the different steps. “context”is the original context, “intermediate\_context”is the context with the answer masked by 1’s. This is translated to produce “translated\_intermediate\_context”. The mask is then replaced by the original answer to form the “reconstructed\_translated\_context”. The details of this process are present in the notebook Augmentation\_wip.ipynb, and are relatively straight forward.

Text

Description automatically generated

## Backtranslation Implementation

Backtranslation was again implemented by using HuggingFace models.

We used the Marian-MT model to perform the work of backtranslation:

Text

Description automatically generated

Text

Description automatically generated

Finally, back-translate is implemented as follows:

Text

Description automatically generated

## Memory leak in Marian-MT model.

The task of translation using these modules is essentially performed by the following lines of code:

encoded = tokenizer.prepare\_seq2seq\_batch(textarray, return\_tensors='pt').to(torch\_device) translated = model.generate(\*\*encoded)

translated\_texts = tokenizer.batch\_decode(translated, skip\_special\_tokens=True)

What we find that although *textarray* is an input parameter to the model, and once the function to prepare\_seq2seq\_batch has returned, the model should not have any references to *textarray.*

However, we find that if *textarray* is still in use after the model has returned, the model starts leaking tensors from inside.

We displayed the memory usage after calling the backtranslate API and found that it leaked a lot of tensors, as shown in the second row below:

A picture containing graphical user interface

Description automatically generated

We didn’t have the time to debug this issue, but we found a work-around. The work-around was to pass a copy of a string to the model, instead of the original string. The copy of the string would be discarded as soon as we were done with it. Our guess, however, is that some sort of circular reference was created within the MarianMT library which led to the leak.

In the code below, line 39 does the work of copying the string:

Text

Description automatically generated

After implementing out fix, no tensors were leaked.

Graphical user interface, application

Description automatically generated

## Implementation of random-swap and random-deletion

The implementations of random-swap and random-deletion were quite straight forward, and are self explanatory.

## Results with Data Augmentation

Data augmentation roughly doubled the size of the data. However, when we ran with the augmented data, we found that the model actually performed worse than with the original data. We found that the best F1 was 69.45 (as opposed to 70.67 with the base dataset).

[12.31.21 09:11:08] Epoch: 0

[12.31.21 09:15:23] Eval F1: 07.53, EM: 00.05

[12.31.21 09:45:51] Eval F1: 59.27, EM: 43.23

[12.31.21 10:16:18] Eval F1: 63.09, EM: 47.17

[12.31.21 10:46:44] Eval F1: 65.45, EM: 49.50

[12.31.21 11:17:12] Eval F1: 66.30, EM: 50.13

[12.31.21 11:47:40] Eval F1: 67.12, EM: 51.30

[12.31.21 12:18:07] Eval F1: 68.42, EM: 52.06

100% 488383/488383 [3:09:44<00:00, 42.90it/s, NLL=0.876, epoch=0]

[12.31.21 12:20:52] Epoch: 1

[12.31.21 12:48:49] Eval F1: 68.03, EM: 52.64

[12.31.21 13:19:35] Eval F1: 68.30, EM: 52.32

[12.31.21 13:50:18] Eval F1: 68.40, EM: 52.59

[12.31.21 14:21:03] Eval F1: 68.62, EM: 52.92

[12.31.21 14:51:48] Eval F1: 68.86, EM: 53.33

[12.31.21 15:22:33] Eval F1: 68.84, EM: 53.19

100% 488383/488383 [3:07:13<00:00, 43.48it/s, NLL=0.762, epoch=1]

[12.31.21 15:28:05] Epoch: 2

[12.31.21 15:53:14] Eval F1: 69.02, EM: 53.26

[12.31.21 16:23:57] Eval F1: 68.29, EM: 52.61

[12.31.21 16:54:39] Eval F1: 68.59, EM: 52.95

[12.31.21 17:25:20] Eval F1: 68.55, EM: 52.64

[12.31.21 17:56:00] Eval F1: 68.32, EM: 52.89

[12.31.21 18:26:40] Eval F1: 69.43, EM: 53.53

100% 488383/488383 [3:06:52<00:00, 43.56it/s, NLL=0.512, epoch=2]

[12.31.21 18:34:57] Epoch: 3

[12.31.21 18:57:18] Eval F1: 68.87, EM: 53.16

[12.31.21 19:27:58] Eval F1: 68.84, EM: 52.89

[12.31.21 19:58:36] Eval F1: 68.47, EM: 52.42

[12.31.21 20:29:16] Eval F1: 68.41, EM: 52.68

[12.31.21 20:59:55] Eval F1: 68.35, EM: 52.35

[12.31.21 21:30:36] Eval F1: 67.80, EM: 51.83

100% 488383/488383 [3:06:41<00:00, 43.60it/s, NLL=0.331, epoch=3]

[12.31.21 21:41:39] Epoch: 4

[12.31.21 22:01:10] Eval F1: 68.11, EM: 51.90

[12.31.21 22:31:42] Eval F1: 67.75, EM: 51.63

[12.31.21 23:02:13] Eval F1: 68.13, EM: 52.02

[12.31.21 23:32:44] Eval F1: 67.87, EM: 51.71

[01.01.22 00:03:09] Eval F1: 67.72, EM: 51.61

[01.01.22 00:33:32] Eval F1: 68.11, EM: 52.04

100% 488383/488383 [3:05:35<00:00, 43.86it/s, NLL=0.0664, epoch=4]

[01.01.22 00:51:30] Final eval results after epoch: F1: 68.05, EM: 51.94

[01.01.22 00:51:30] After epoch, best scores so far: OrderedDict([('F1', 69.42591031886586), ('EM', 53.530306321963934)])

# Slanted Triangle Learning Rate

Following the approach presented by Ma and Yarats[[1]](#footnote-1), and Howard and Ruder[[2]](#footnote-2) we implemented slanted triangle learning rates using warmup and learning rate decay. We experimented with linear and exponential warmup, and a cosine learning rate scheduler.

We used the pytorch\_warmup library to vary the learning rate.

Text

Description automatically generated

## Results with Slanted Triangle Learning Rate

However, this model completely failed to converge, and we did not experiment too much with the different parameters to see if we could make this work.

Since we did not experiment too much with this strategy, the results have not been presented, as the efforts were abandoned early on to save time for other more promising methods.

# Gradual Unfreezing

Inspired by Howard and Ruder[[3]](#footnote-3), we implemented gradual unfreezing of layers. We followed the following unfreezing schedule:



Text

Description automatically generated

## Results with Gradual Unfreezing

We ran this model with the augmented dataset, and this performed better than all past methods that we used, achieving an F1 of 71.63 and EM of 55.32.

[01.01.22 12:55:48] Epoch: 0

0% 0/488383 [00:00<?, ?it/s]

Iteration: 0 unfreezing from distilbert.transformer.layer.5.attention.q\_lin.weight Unfroze 18 layers

[01.01.22 13:00:05] Eval F1: 07.54, EM: 00.06

7% 32000/488383 [08:59<1:06:58, 113.57it/s, NLL=2.06, epoch=0]

Iteration: 1000 unfreezing from distilbert.transformer.layer.4.attention.q\_lin.weight Unfroze 34 layers

[01.01.22 13:17:43] Eval F1: 41.07, EM: 26.54

26% 128000/488383 [30:34<1:04:54, 92.53it/s, NLL=1.41, epoch=0]

Iteration: 4000 unfreezing from distilbert.transformer.layer.3.attention.q\_lin.weight Unfroze 50 layers

[01.01.22 13:37:31] Eval F1: 52.52, EM: 36.42

39% 192000/488383 [48:34<1:03:21, 77.97it/s, NLL=1.54, epoch=0]

Iteration: 6000 unfreezing from distilbert.transformer.layer.2.attention.q\_lin.weight Unfroze 66 layers

[01.01.22 14:00:36] Eval F1: 60.56, EM: 43.99

52% 256000/488383 [1:08:46<57:48, 67.00it/s, NLL=1.21, epoch=0]

Iteration: 8000 unfreezing from distilbert.transformer.layer.1.attention.q\_lin.weight Unfroze 82 layers

66% 320000/488383 [1:26:51<47:26, 59.15it/s, NLL=1.1, epoch=0]

Iteration: 10000 unfreezing from distilbert.transformer.layer.0.attention.q\_lin.weight Unfroze 98 layers

[01.01.22 14:26:57] Eval F1: 64.60, EM: 48.34

79% 384000/488383 [1:51:23<33:03, 52.63it/s, NLL=0.957, epoch=0]

Iteration: 12000 unfreezing all all Unfroze 102 layers

[01.01.22 14:56:43] Eval F1: 66.22, EM: 49.77

[01.01.22 15:27:02] Eval F1: 67.90, EM: 51.53

100% 488383/488383 [2:33:57<00:00, 52.87it/s, NLL=0.768, epoch=0]

[01.01.22 15:29:46] Epoch: 1

[01.01.22 15:57:37] Eval F1: 68.29, EM: 52.15

[01.01.22 16:28:15] Eval F1: 69.08, EM: 52.55

[01.01.22 16:58:50] Eval F1: 69.41, EM: 53.13

[01.01.22 17:29:26] Eval F1: 69.75, EM: 53.50

[01.01.22 18:00:02] Eval F1: 70.31, EM: 53.87

[01.01.22 18:30:37] Eval F1: 69.98, EM: 54.04

100% 488383/488383 [3:06:21<00:00, 43.68it/s, NLL=0.944, epoch=1]

[01.01.22 18:36:07] Epoch: 2

[01.01.22 19:01:09] Eval F1: 70.46, EM: 54.34

[01.01.22 19:31:42] Eval F1: 70.74, EM: 54.59

[01.01.22 20:02:16] Eval F1: 70.20, EM: 54.07

[01.01.22 20:32:49] Eval F1: 70.64, EM: 54.50

[01.01.22 21:03:26] Eval F1: 70.69, EM: 54.70

[01.01.22 21:34:03] Eval F1: 71.30, EM: 54.89

100% 488383/488383 [3:06:12<00:00, 43.71it/s, NLL=0.755, epoch=2]

[01.01.22 21:42:20] Epoch: 3

[01.01.22 22:04:40] Eval F1: 71.29, EM: 54.88

[01.01.22 22:35:15] Eval F1: 71.29, EM: 55.02

[01.01.22 23:05:48] Eval F1: 71.26, EM: 55.24

[01.01.22 23:36:22] Eval F1: 70.89, EM: 54.71

[01.02.22 00:06:54] Eval F1: 71.54, EM: 54.97

[01.02.22 00:37:27] Eval F1: 71.43, EM: 55.04

100% 488383/488383 [3:06:06<00:00, 43.74it/s, NLL=0.236, epoch=3]

[01.02.22 00:52:44] Final eval results after epoch: F1: 71.63, EM: 55.32

We stopped training after 4 epochs because of time constraints, but we could see that the model was still improving its accuracy. Future work here would involve training further, more than 4 epochs and then finetuning further by decreasing the learning rate as well. Another thing to experiment with would be a smaller batch size in later iterations.

# Comparison of the three approaches

We plot the outcome of all the three approaches. In the below plots, the F1 and EM scores are plotted against the iterations. The vertical lines indicate epochs, and at which iteration they end.

The “augmented” run does not have gradual unfreezing, and the “gradual\_unfreeze” line includes augmented data.

We clearly see that the both the runs without gradual unfreezing plateau quite early, and after the first epoch, their performance starts dropping.

We also see that the run with data augmentation and gradual unfreezing of layers continues to improve till epoch 4, at which point we stop testing.

Chart

Description automatically generated

Chart, histogram

Description automatically generated

# Future work

Gradual unfreezing along with augmentation has shown promise and continues to scale till epoch 4. We stopped at epoch 4 because of time constraints, but future work could involve training beyond epoch 4 till the time we hit a plateau, and then train with a lower learning rate.

We could not get slanted triangle learning rate to work, but the fact that it completely failed to converge when we tried to modify the learning rate across iterations indicates that this does have an effect. More values should be tried to see if we can find a set of parameters where we can not only get the model to converge, but to actually improve the performance.

We used a very similar language like French for backtranslation, but it will be interesting to compare backtranslation across other languages that are not similar to English.

1. Ma, J., & Yarats, D. (2019). On the adequacy of untuned warmup for adaptive optimization. arXiv preprint arXiv:1910.04209, 7. [↑](#footnote-ref-1)
2. Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146. [↑](#footnote-ref-2)
3. Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. arXiv preprint arXiv:1801.06146. [↑](#footnote-ref-3)