Report

Contents

1. Name

Simon Raviv

2. Paper Information

I have chosen **A Decomposable Attention Model for Natural Language Inference**, by Parikh et al. '16, the base model, without intra-sentence attention.

In the last part of the course, we learnt about attention and transformers. In recent years, models based on those techniques are some of current state of the art and I wanted to get a deeper understanding about those techniques.

In addition, this paper was able to get relatively high accuracy with number of parameters (380K) lower almost by order of magnitude compared to the other papers, which were using millions of parameters. I was curious to understand the way they did it, as I think there is place for models with low number of parameters. The use cases of such models can be low latency inference and low resources edge devices.

3. System Information

Those are the relevant software/hardware components I have used for development and training of the model.

3.1. Hardware

- CPU: Intel(R) Xeon(R) Gold 6336Y CPU @ 2.40GHz with 48 physical cores
- GPU: NVIDIA RTX™ A6000 48 GB GDDR6
- RAM: 256GB

3.2. Software

- OS: Ubuntu 20.04.4 LTS
- GPU Driver: 515.48.07
- CUDA: 11.7
- Python: 3.8.10
- PyTorch: 1.11.0+cu113

4. The Code

I have written generic application in PyTorch with all the features controllable by the command line, see the detailed information regarding the usage in the README.

The original paper was implemented in TensorFlow, but the code was not published. There could be differences in the results due to differences in the libraries themself.

5. The Method

This is brief description of the method presented in the paper, more detailed explanation including the full formulation can be seen in the paper itself. See the model architecture in the model architecture section.

5.1. Preliminary Definitions

- Let $\boldsymbol{a} = (a_1, ., ., , a_{l_a})$ and $\boldsymbol{b} = (b_1, ., ., , b_{l_b})$ be the two input sentences
- l_a and l_b, the sentences length respectively
- Each $a_i, b_j \in \mathbb{R}^d$ is a word embedding vector of dimension d
- Each sentence is prepended with a "NULL" token
- Training data in a form of labeled pairs: $\{\boldsymbol{a}^{(n)}, \boldsymbol{b}^{(n)}, \boldsymbol{y}^{(n)}\}_{n=1}^{N}$
  - $\boldsymbol{y}^{(n)} = \left(y_1^{(n)}, \ldots, y_c^{(n)}\right)$ is an indicator vector encoding the label
  - C is the number of output classes

At test time, the model receives a pair of sentences $(\boldsymbol{a}, \boldsymbol{b})$ and the goal is to predict the correct label $\boldsymbol{y}$.

5.2. Pictorial Overview

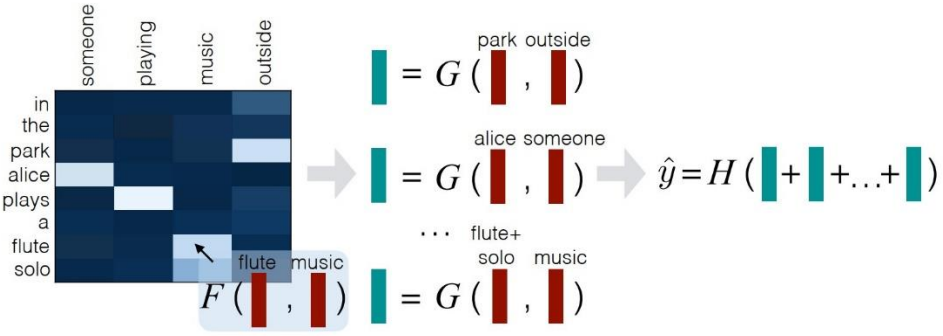

Figure 1: Pictorial overview of the approach, showing the *Attend* (left), *Compare* (center) and *Aggregate* (right) steps.

5.3. Method Description

5.3.1. Input Representation

Let $\bar{\boldsymbol{a}} = (\bar{a}_1, ., ., , \bar{a}_{l_a})$ and $\bar{\boldsymbol{b}} = (\bar{b}_1, ., ., , \bar{b}_{l_b})$ denote the input representation of each fragment that is fed to subsequent steps of the algorithm. The vanilla version of the model, which I used, simply defines $\bar{\boldsymbol{a}} := \boldsymbol{a}$ and $\bar{\boldsymbol{b}} := \boldsymbol{b}$. With this input representation, the model does not make use of word order.

The core model consists of the following three components (see Figure 1), which are trained jointly.

5.3.2. Attend

First, soft-align the elements of $\bar{\boldsymbol{a}}$ and $\bar{\boldsymbol{b}}$ using a variant of neural attention and decompose the problem into the comparison of aligned subphrases.

5.3.3. Compare

Second, separately compare each aligned subphrase to produce a set of vectors $\{V_{1,i}\}_{i=1}^{l_a}$ for $\boldsymbol{a}$ and $\{V_{2,j}\}_{j=1}^{l_b}$ for $\boldsymbol{b}$. Each $V_{1,i}$ is a nonlinear combination of a_i and its (softly) aligned subphrase in $\boldsymbol{b}$ (and analogously for $V_{2,j}$).

5.3.4. Aggregate

Finally, aggregate the sets $\{V_{1,i}\}_{i=1}^{l_a}$ and $\{V_{2,j}\}_{j=1}^{l_b}$ from the previous step and use the result to predict the label $\hat{y}$.

6. Model Architecture

6.1. Architecture

```
DecomposableAttentionModel(
  (input_encoder): InputEncoder(
    (embedding): Embedding(39079, 300)
    (input_linear): Linear(in_features=300, out_features=300, bias=False)
  )
  (intra_attention): Attention(
    (f_mlp): MLP(
      (fc): Sequential(
        (0): Dropout(p=0.2, inplace=False)
        (1): Linear(in_features=300, out_features=300, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.2, inplace=False)
        (4): Linear(in_features=300, out_features=300, bias=True)
        (5): ReLU()
      )
    )
    (g_mlp): MLP(
      (fc): Sequential(
        (0): Dropout(p=0.2, inplace=False)
        (1): Linear(in_features=600, out_features=300, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.2, inplace=False)
        (4): Linear(in_features=300, out_features=300, bias=True)
        (5): ReLU()
      )
    )
    (h_mlp): MLP(
      (fc): Sequential(
        (0): Dropout(p=0.2, inplace=False)
        (1): Linear(in_features=600, out_features=300, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.2, inplace=False)
        (4): Linear(in_features=300, out_features=300, bias=True)
        (5): ReLU()
      )
    )
    (linear_output): Linear(in_features=300, out_features=3, bias=True)
    (softmax): LogSoftmax(dim=1)
  )
)
```

6.2. Flow

Figure 2: Model main components flow.

7. Dataset

The data used to evaluate the proposed approach is the Stanford Natural Language Inference (SNLI) dataset (Bowman et al., 2015). Given a sentences pair ($\boldsymbol{a}$, $\boldsymbol{b}$), the task is to predict whether $\boldsymbol{b}$ is *entailed* by $\boldsymbol{a}$, $\boldsymbol{b}$ *contradicts* $\boldsymbol{a}$, or whether their relationship is *neutral*.

The application implemented supports the process below through the command line, see README for usage information.

7.1. Data Preprocessing

The dataset contains the following number of sentences:

- Train: 549,366
- Dev: 9,841
- Test: 9,823

The data is loaded from SNLI dataset, then it indexed for word and label indexes. After that the data was batched to a given batch size and finally saved as HDF5 files for train, dev and test sets to be loaded later in the training/testing process with all the information needed, besides the data itself.

7.2. Word Embedding Preprocessing

The paper used 300-dimensional GloVe with 840 billion parameters embeddings (Pennington et al., 2014) to represent words. Each embedding vector was normalized to have l_2 norm.

Out-of-vocabulary (OOV) words are hashed to one of 100 random embeddings each initialized to mean 0 and standard deviation 1.

Finally, the embedding of words exists in the dataset were saved as HDF5 file for training/test process.

8. Missing Implementation Details Handling

In addition to hyper-parameters issues described in the results replication section, there were some implementation details that were missing and didn't work out of the box. My first implementation was missing them, after the addition, the results did improve.

Below I describe some of them.

8.1. Embedding Projection

The authors mentioned they did projection of the embedding vectors from 300 dimensional vectors to 200 dimensional. However, they didn't mention the method used to do this projection. Due to that I have chosen to remain with the original 300-dimensional vectors, as the projection method itself can have not neglectable impact on the results, especially since it mentioned the projection weights are trained as well.

I think this is why they chose the hidden size layer of the linear and MLP layer to be 200. I did experiments with 200 and 300 dimensions for the hidden size and using 300 for the hidden size showed better results. I think it's due to the fact my embeddings were 300 dimensional and not 200 dimensional as in the paper after the projection.

8.2. Gradients Clipping

After my first implementation, the results weren't good. I have tried to look on different papers implementation to see if there are common implementation details used there. This paper didn't discuss gradient sizes and gradient exploding issue. I have noticed that many of them used gradients clipping. After adding gradients clipping, there training became reasonable and stable. I have used a common value I saw in use, my max normal size for the gradient was 5.

8.3. Optimizer

The paper used Adagrad optimizer. I have used one instance of PyTorch Adagrad optimizer with the parameters reported in the paper to the whole model (both input encoder and attention). The results weren't too bad, but it was a bit too far from the results reported in the paper.

After an investigation of how the optimizer works, I have noticed the optimizer includes parameters of state sum that affects all parameters of the model, see here. I realized it would be better to have 2 optimizer instances, one for each sun model, input encoder and another for the attention layer.

This indeed improved the results by non-neglectable factor.

9. Results Analysis

9.1. Paper Results

Train Acc	Test Acc
89.9	86.3

Table 1: Train/test accuracies % on the SNLI dataset - paper results.

N	E	C
83.6	91.3	85.8

Table 2: Breakdown of accuracy % with respect to classes on SNLI development set. **N**=neutral, **E**=entailment, **C**=contradiction - paper results.

Notes:

- The authors didn't supply the accuracy on the overall dev set.
- The authors mentioned they took the model did best on dev dataset, but it's not clear if the train accuracy the present are from the save saved model or on the last model.

9.2. Replicated Results

9.2.1. Results

Train Acc	Dev Acc	Test Acc
86.14	86.1	85.43

Table 3: Train/test accuracies % on the SNLI dataset - replicated results.

N	E	C
81.3	90.53	81.3

Table 4: Breakdown of accuracy % with respect to classes on SNLI development set. **N**=neutral, **E**=entailment, **C**=contradiction - replicated results.

9.2.2. Training Graphs

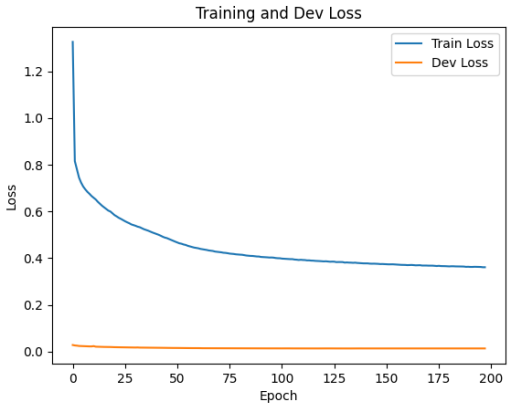

Figure 3: Replicated model training loss on train/dev datasets

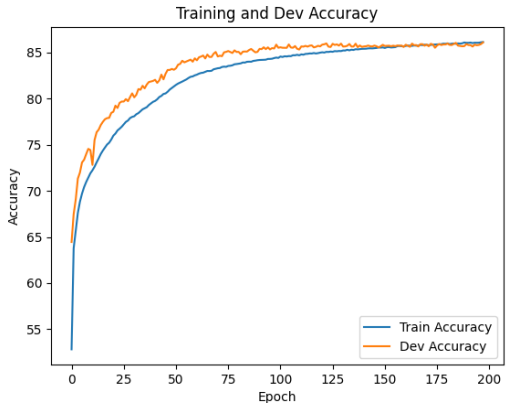

Figure 4: Replicated model training accuracy on train/dev datasets

9.2.3. Training Hyper-Parameters

Parameter	Value
Parameters Gaussian initialization mean, std	0, 0.01
Optimizer	Adagrad
Adagrad initial accumulator value	0
Dataset shuffle at each epoch	True
Epochs	250
Batch size	32
Embedding dimension	300
Number of layers in MLPs	2
MLP hidden dimension	300
Learning rate	0.05
Dropout	0.2
Weight decay	1e-0
Max gradient normal	5

Table 5: Replicated results hype-parameters.

Notes:

- Those are not exactly the parameters mentioned in the paper, since the best results were achieved with different parameters, see missing implementation details handling section and replication efforts section.
- The best models saved in each epoch, the model chosen was the one with the best dev accuracy, which was achieved in epoch 198.

9.3. Paper Results vs Replicated Results

Model	Train Acc	Test Acc
Paper	89.9	86.3
Replicated	86.1	85.4

Table 6: Train/test accuracies % on the SNLI dataset - paper vs. replicated results.

Model	N	E	C
Paper	83.6	91.3	85.8
Replicated	81.3	90.5	81.3

Table 7: Breakdown of accuracy % with respect to classes on SNLI development set.
N=neutral, **E**=entailment, **C**=contradiction - paper vs. replicated results.

I think the important results here are the dev and test accuracies. The test results achieved are very close to the paper, the difference is ~0.9[%], this could be due to multiple reasons described in missing implementation details handling section and in results replication efforts section.

9. Replication Efforts

First, there were several implementation details affected results replication, they are described in missing implementation details handling section.

In addition, as mentioned in training hyper-parameters section, the best results weren't achieved using the parameters mentioned in the paper. The authors of the paper mention the parameters achieved chosen in hyper-parameters tuning process, but due to missing information and implementation differences discussed in this report, I had to change those parameters.

I have conducted several experiments to find the parameters that achieved the results in previous section. In the next sections I describe some of them.

9.1. Paper Parameters – Experiment 1

The first experiment I have conducted was with the original parameters mentioned in the paper. The experiment duration was 12:09:07 hours.

Parameter	Value
Parameters Gaussian initialization mean, std	0, 0.01
Optimizer	Adagrad
Adagrad initial accumulator value	0.1
Dataset shuffle at each epoch	True
Epochs	250
Batch size	16[1]
Embedding dimension	300[2]
Number of layers in MLPs	2
MLP hidden dimension	200
Learning rate	0.05
Dropout	0.2
Weight decay	0[3]
Max gradient normal	5[3]

Table 8: Paper hype-parameters, see notes.

Train Acc	Dev Acc	Test Acc
79.63	80.15	79.64

Table 9: Train/test accuracies % on the SNLI dataset - replicated results with original paper hyper-parameters.

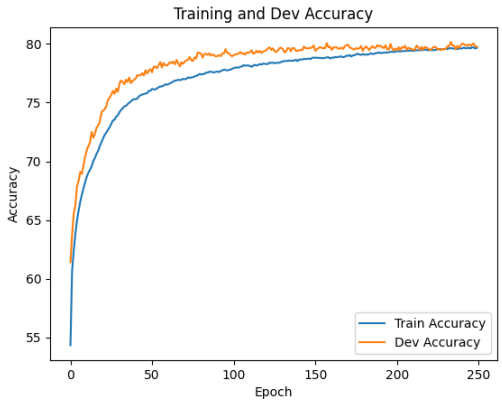

Figure 5: Paper parameters model training accuracy on train/dev datasets.

Notes:

1. The results in the paper used batch size 4, but it was mentioned they also did runs with 16 and 32 and the results were stable.
2. As mentioned in embedding projection section, I have used the full embedding dimension.
3. The parameters weren't mentioned in the paper, used common default.

9.2. Experiment 2

In the second experiment I have updated two parameters, see the bolded parameters in table 10 below. This improved the accuracy on test set by 5.19[%], see table 11 below.

The experiment duration was 12:00:48 hours.

Parameter	Value
Parameters Gaussian initialization mean, std	0, 0.01
Optimizer	Adagrad
Adagrad initial accumulator value	**0**
Dataset shuffle at each epoch	True
Epochs	250
Batch size	16[1]
Embedding dimension	300[2]
Number of layers in MLPs	2
MLP hidden dimension	200
Learning rate	0.05
Dropout	0.2
Weight decay	**0.00001[3]**
Max gradient normal	5[3]

Table 10: Experiment 2 hype-parameters, see notes in table 8. Bolded parameters are changed from previous experiment.

Train Acc	Dev Acc	Test Acc
84.2	85.26	84.83

Table 11: Train/test accuracies % on the SNLI dataset - replicated results with experiment 2 hyper-parameters.

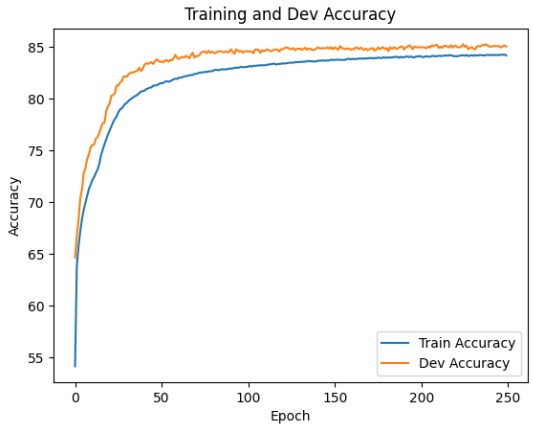

Figure 6: Experiment 2 model training accuracy on train/dev datasets.

9.3. Experiment 3

In the third experiment I thought above the dimension of MLP hidden size. As mentioned in embedding projection section, I have chosen to remain with the original size of embedding dimeson, 300. Thus, due to the input being 100 dimensions bigger than the hidden layers sizes, I chose to increase the dimension of the hidden layers to 300 as well. The motivation is the larger input dimension in my replicated model.

This improved the accuracy on test set by 0.46[%], see table 13 below.

The experiment duration was 11:41:47 hours.

Parameter	Value
Parameters Gaussian initialization mean, std	0, 0.01
Optimizer	Adagrad
Adagrad initial accumulator value	0
Dataset shuffle at each epoch	True
Epochs	250
Batch size	16[1]
Embedding dimension	300[2]
Number of layers in MLPs	2
MLP hidden dimension	**300**
Learning rate	0.05
Dropout	0.2
Weight decay	0.00001[3]
Max gradient normal	5[3]

Table 12: Experiment 3 hype-parameters, see notes in table 8. Bolded parameters are changed from previous experiment.

Train Acc	Dev Acc	Test Acc
85.71	86.02	85.29

Table 13: Train/test accuracies % on the SNLI dataset - replicated results with experiment 3 hyper-parameters.

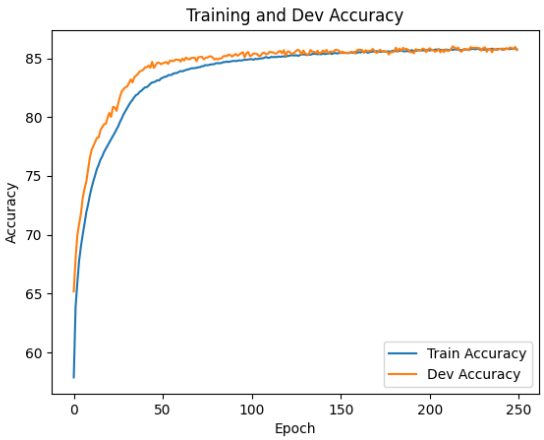

Figure 7: Experiment 3 model training accuracy on train/dev datasets.

9.4. Final experiment

I have conducted some more experiments; I think the presented above are enough to get the picture of my process of replicating the results.

The final submitted model was using the parameters in experiment 3 with batch size of 32. This gave slightly better results. Those are the results presented in replicated results section. The final experiment took 06:21:24 hours.

10. My Implementation Results Replication

My code is reproduceable, see README on how to reproduce my training process exactly.

In addition, I supply a way to download my model and run evaluation on any test set, i.e., dev/test, see instructions in the README.

As noted in the system information section, I run the training on a NVIDIA GPU. My code is cross platform between CPU/GPU, so the uploaded model could also be run on CPU, but in this case the results will be a bit different in about ~0.1 [%] due to this. Even on my platform, running inference test on the same system give slightly different results between CPU/GPU.

Finally, my model training process with all details available in the training log I have added to the submission.