Project 3 - Semantic Code Search

Professor Pantelis Monogioudis

Zhu, Haoran - hz1922

Li, Qi - ql1045

Zhao, Jingyi - jz2216

Project Goal

In this project, we use the CodeSearchNet Corpus and participate in the corresponding challenge. We focus only on Python language and the associated dataset contains about 0.5 million pairs of function-documentation pairs and about another 1.1 million functions without an associated documentation. We submit our Normalized Discounted Cumulative Gain (NDCG) score for only the human annotated examples.

We complete this project as following:

1. Read *"CodeSearchNet Challenge Evaluating the State of Semantic Code Search"*
2. Do research on all 5 models implemented on the *CodeSearchNet.*
3. Follow the setup instructions
4. Implement all 5 baseline models
5. Compare and discuss the results of the 5 models
6. Improve the self attention model with pre-trained BERT-Tiny(2/128) by Google

Algorithms In Baseline Models

Neural Bag of Words

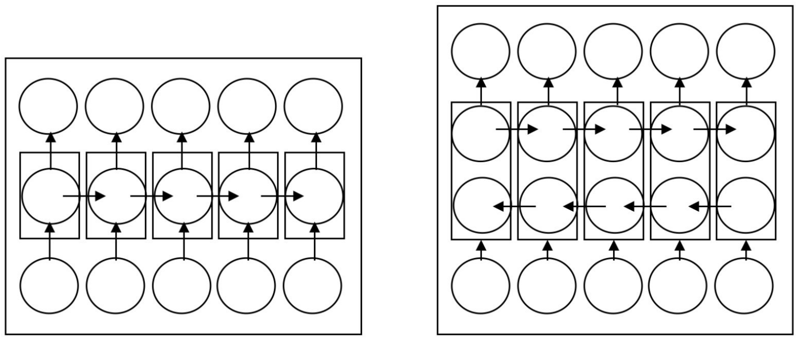
A bag-of-words model is a way of extracting features from text for use in modeling. It is a representation of text that describes the occurrence of words within a document.

The **Neural Bag-of-Words (NBOW)** model from *“A Convolutional Neural Network for Modelling Sentences”* is a fully connected network which maps text X, a sequence of words, to one of k out- put labels. The NBOW model has d dimensional word vectors for each word in the chosen task vocabulary. For the words w ∈ X, corresponding word vectors vw are looked up and a hidden vector representation z is obtained as an average of the input word vectors $z = \frac{1}{|X|} \sum_{w \in X} v_w$. The average vector z is then fed to a fully con- nected layer to estimate probabilities for the out- put labels as: $\hat{y} = softmax\,(W_l z + b\,)$. For text classification tasks, the NBOW model is trained to minimise the categorical cross-entropy loss using a stochastic gradient descent al- gorithm. Additional fully connected layers can be added into the NBOW model to form Deep Aver- aging Networks (DAN).

Bidirectional RNN models

Implement the GRU cell from *“On the Properties of Neural Machine Translation: Encoder–Decoder Approaches”* to summarize the input sequence.

Bidirectional Recurrent Neural Networks (BRNN) connect two hidden layers of opposite directions to the same output.



(a) (b)

Structure overview

(a) unidirectional RNN
(b) bidirectional RNN

The principle of BRNN is to split the neurons of a regular RNN into two directions, one for positive time direction (forward states), and another for negative time direction (backward states). Those two states' output are not connected to inputs of the opposite direction states. The general structure of RNN and BRNN can be depicted in the right diagram. By using two time directions, input information from the past (backwards) and future (forward) of the current time frame can be used unlike standard RNN which requires the delays for including future information.

Gated Recurrent Unit(GRU): In GRU, there is no explicit memory unit. Memory unit is combined along with the network. There is no forget gate and update gate in GRU. They are both combined together and thus the number of parameters are reduced.

In CodeSearch, the baseline model default set two rnn layers, LSTM cell, weighted_mean pool mode and run as biRNN. There is a confusion when implementing the baseline model: from the paper, the author used GRUcell, but in the model, the developer used LSTMcell. In the following experiment, we chose to use the developer's method.

1D Convolutional Neural Network

Implement the algorithm from *"Convolutional Neural Networks for Sentence Classification"*.

This is a **CNN with one layer of convolution** on top of word vectors obtained from an unsupervised neural language model. Word vectors trained by word2vec.

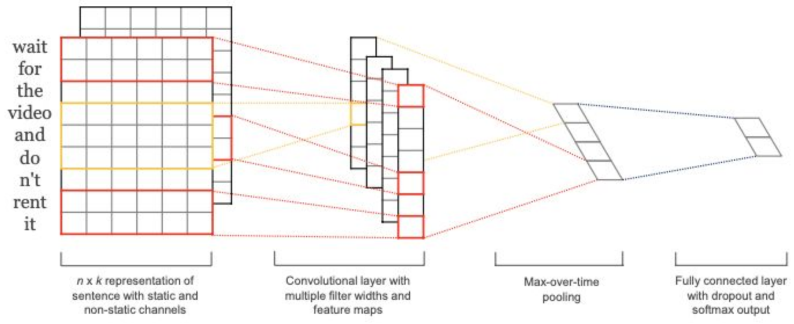

wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 1: Model architecture with two channels for an example sentence.

Model detail:
A sentence of length n is represented as $x_{1:n} = x_1 \oplus x_2 \oplus ... \oplus x_n$. A convolution operation involves a filter w is applied to a window of h words to produce a new feature. A feature ci is generated from a window of words xi:i+h−1 by $c_i = f(w \cdot x_{i:i+h-1} + b)$. Here b ∈ R is a bias term and is a non-linear function such as the hyperbolic tangent. This filter is applied to each possible window of words in the sentence {x1:h, x2:h+1, . . . , xn−h+1:n} to produce a feature map c = [c1,c2,...,cn−h+1]. Then apply a max-over-time pooling operation over the feature map and take the maximum value of c.

Regularization: dropout on the penultimate layer with a constraint on l2-norms of the weight vectors. instead of using $y = w \cdot z + b$, for output unit y in forward propagation, dropout uses $y = w \cdot (z \circ r) + b$, where $\circ$ is the element-wise multiplication opera- tor and r is a ‘masking’ vector of Bernoulli random variables with probability p of being 1. Gradients are backpropagated only through the unmasked units.

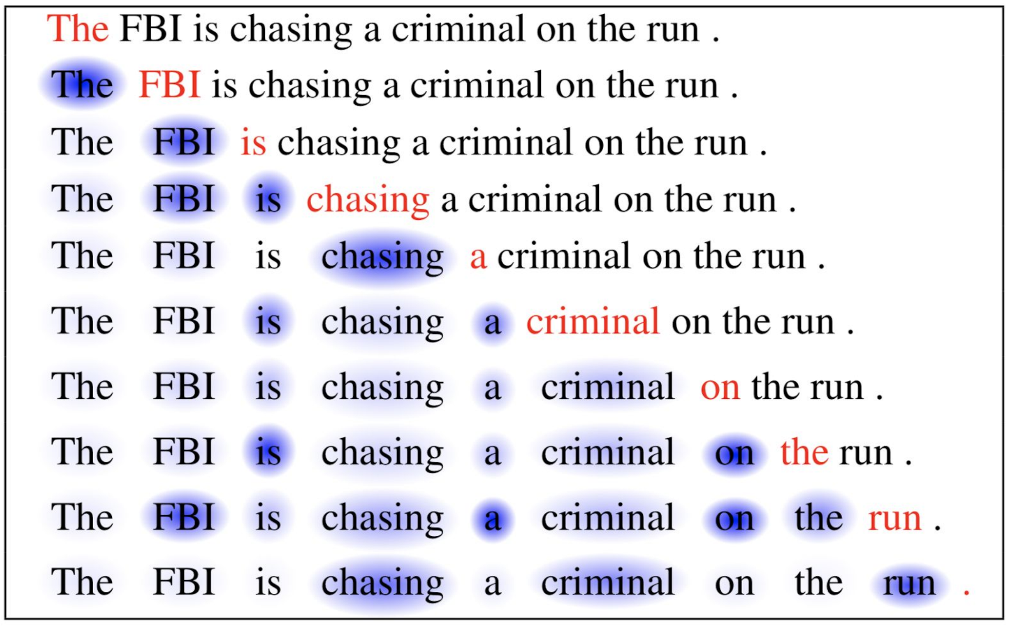
In CodeSearch, we used tanh as activation function, weighted_mean as pool mode, and softmax as output layer.

Self-Attention

According to the great summary by https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html, **self-attention**, also known as **intra-attention**, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence. It has been shown to be very useful in machine reading, abstractive summarization, or image description generation.

It mimics the way human process information: when we see a picture, we often focus on some important regions instead of looking at the whole image; when we see a sentence, we often pay more attention to one part of the whole sentence; The attention mechanism can help us do better correlating inference

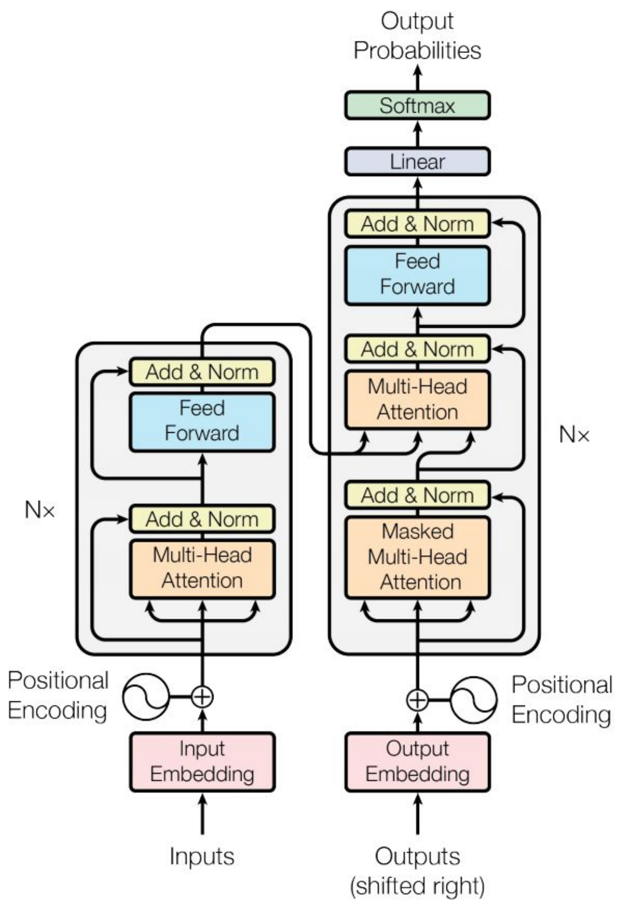
Below is the visualization of attention applied on sentences:


The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .
The FBI is chasing a criminal on the run .

.

(Figure from Cheng et al., 2016, Long Short-Term Memory-Networks for Machine Reading)

In this project, we are using the Transformer model, which uses a Multi-head scaled dot-product attention mechanism(Vaswani, et al., 2017, Attention Is All You Need), below is the architecture:



Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel. The independent attention outputs are simply concatenated and linearly transformed into the expected dimensions.

1D-CNN and Self-Attention Hybrid

The 1D-CNN and self-attention hybrid method is adding self-attention mechanism to the 1D-CNN model described above.

Experiment

We followed the design provided in github/CodeSearchNet. This model uses two encoders to convert query or code into vector representations respectively, after which compute the correlation between the two representations with matrix multiplication, and has a softmax layer as model output.

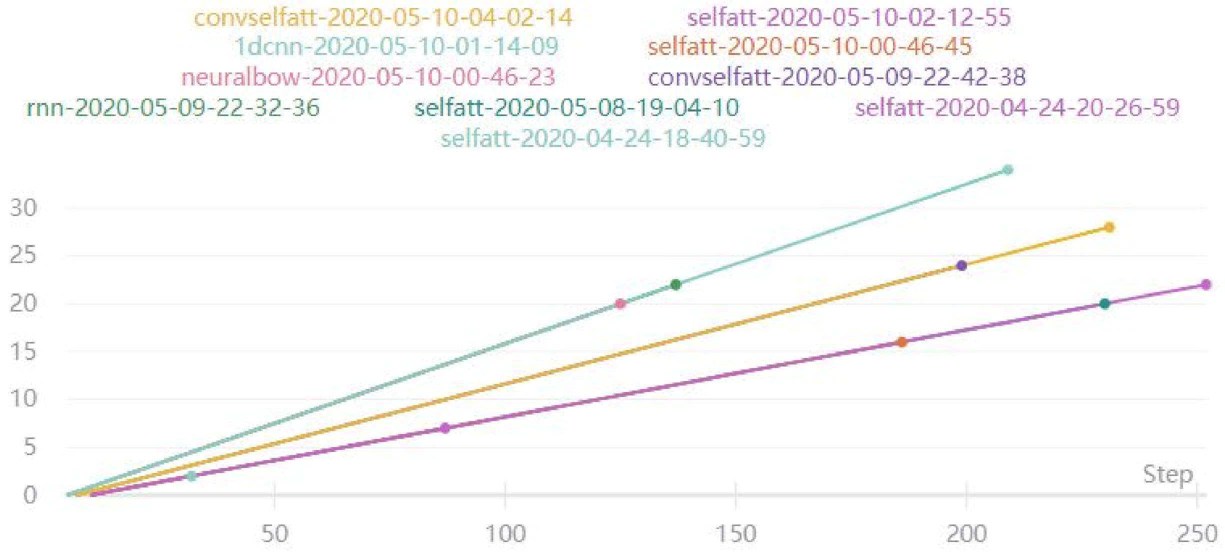
The encoder part is configurable and we have tested 5 different ones including Neural-Bag-Of-Words, RNN, 1D-CNN, Self-Attention (BERT), and a 1D-CNN+Self-Attention Hybrid. For the scope of this project we used only the python dataset, which means this project focuses on a single language instead of multiple ones in the original setting.

The experiments have been conducted on a i9-10900X/64GB/RTX 2080 Ti workstation and a Xeon Gold 5217/192GB/Tesla V100 workstation in case the former one runs out of memory. We found that self-attention(BERT) is a notable resource hungry model and cannot run on a GPU with less than 16GB Graphics RAM. We also noticed that the more Graphics RAM we had, the deeper transformer we could have. The github default implementation used 3 hidden layers of size 128 and experiments showed that this is the maximum size we can have on a standard workstation. Google mentioned that a full BERT model of size 12/768 requires a Cloud TPU with 64GB of RAM. In the future we plan to conduct experiments on a TPU enabled platform.

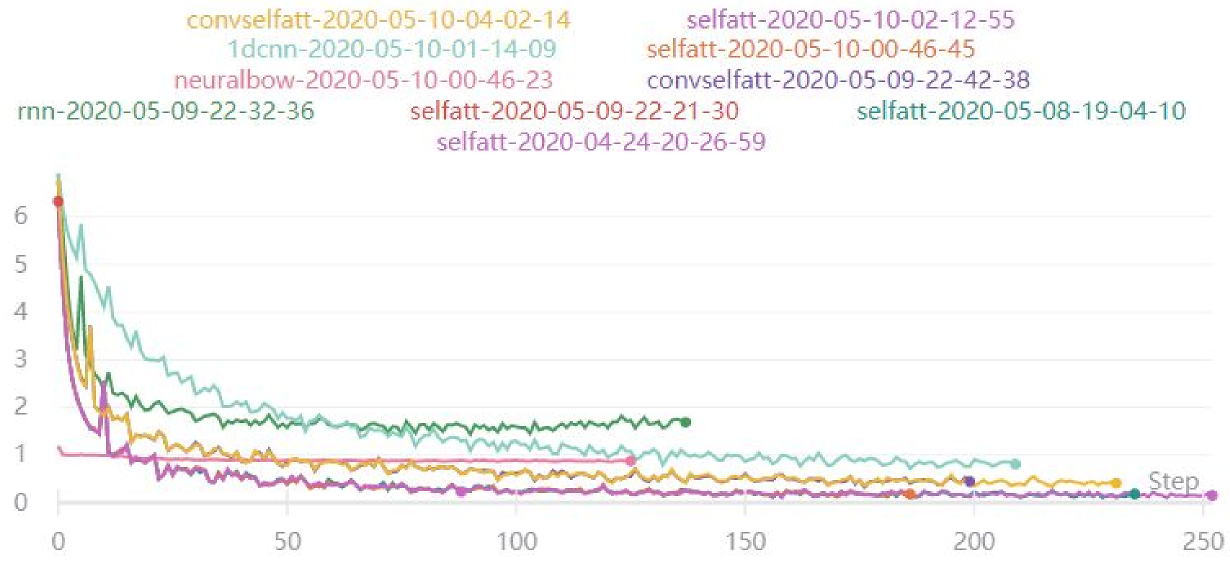
BERT-Tiny is the pretrained transformer presented by Google in the most recent release. We tried to load it into the original model, and we have lowered the transformer depth to 2/128 to match with BERT-Tiny.

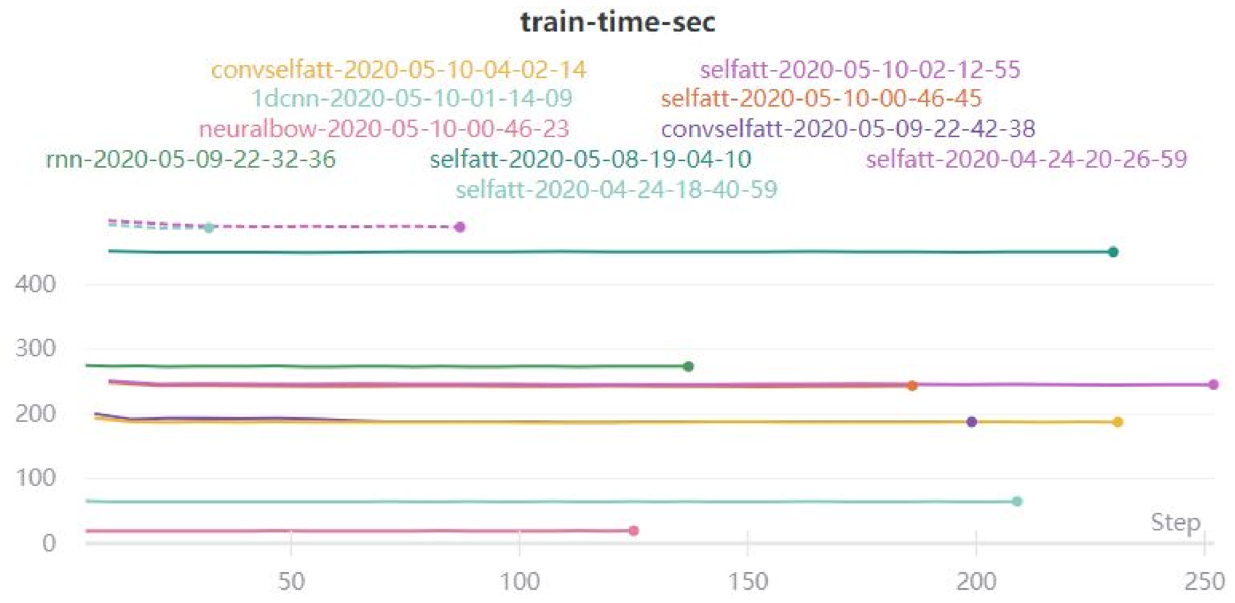
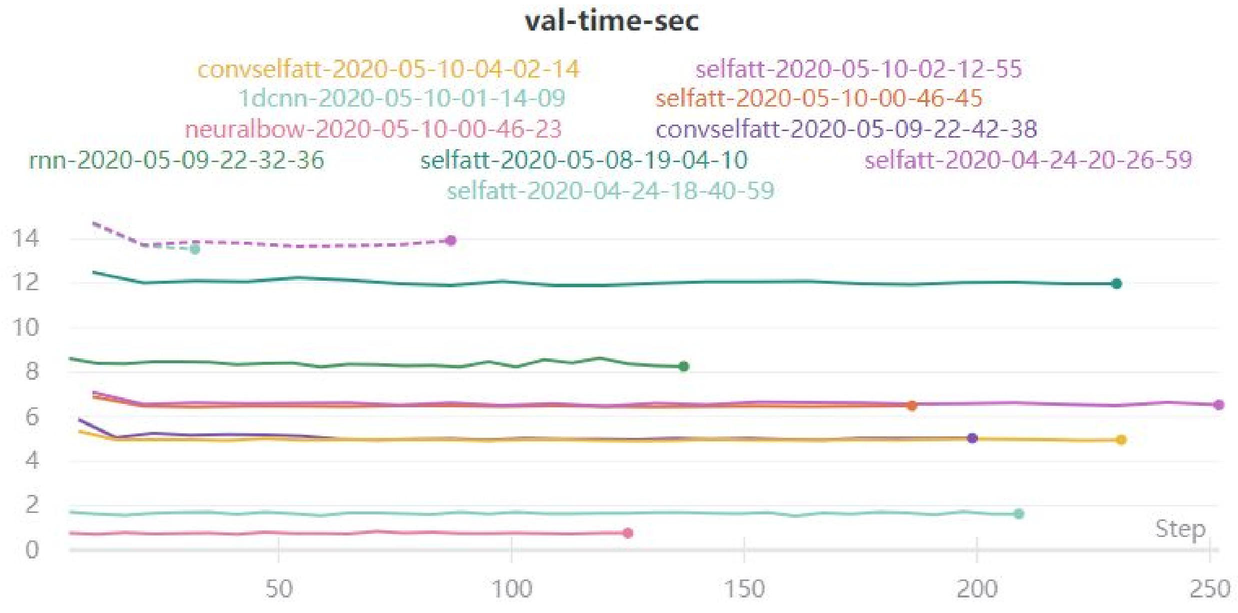
The results of our experiments are shown in the following figures. Each line represents an experiment, and `selfatt-2020-05-10-00-46-45` is the one with pretrained BERT-Tiny loaded.

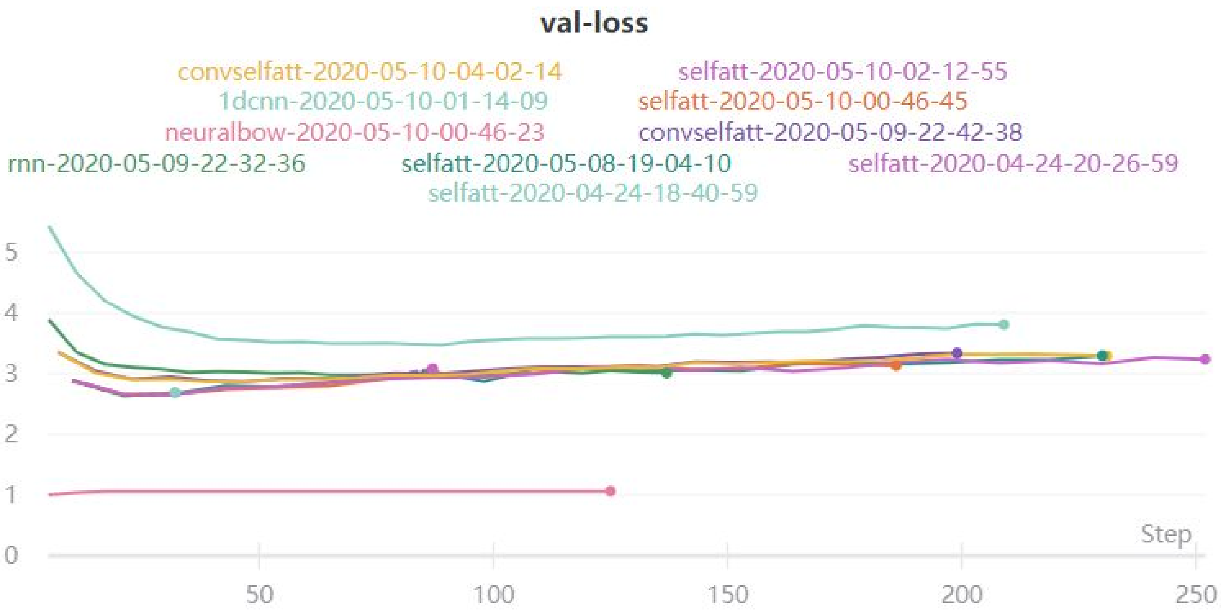
epoch

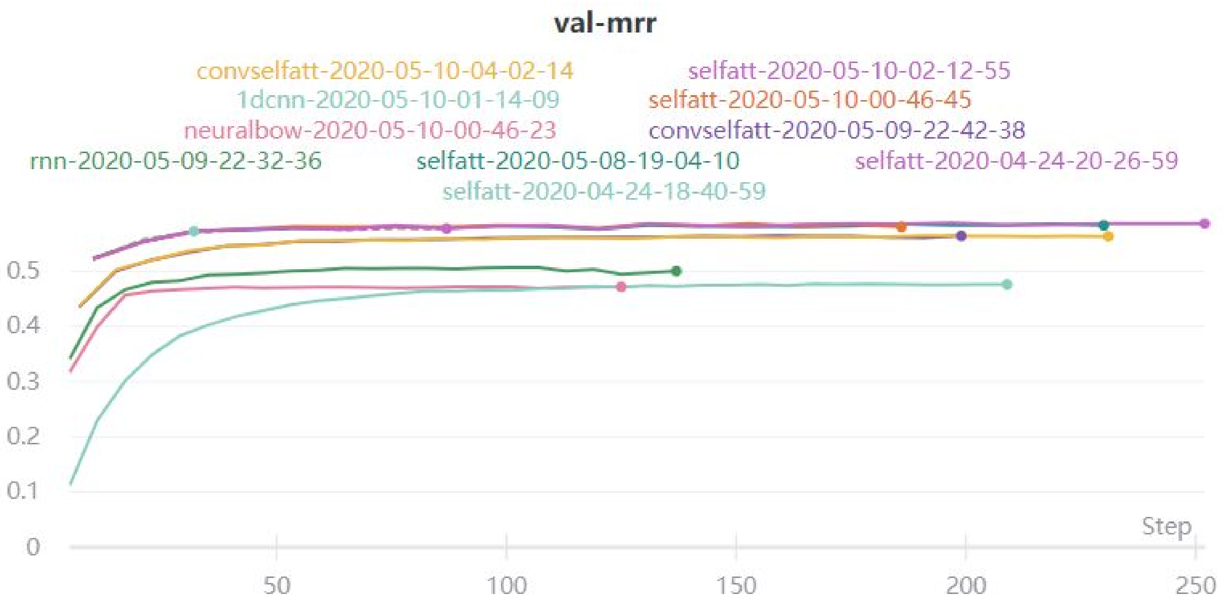


train-loss




train-time-sec
convselfatt-2020-05-10-04-02-14
selfatt-2020-05-10-02-12-55
1dcnn-2020-05-10-01-14-09
selfatt-2020-05-10-00-46-45
neuralbow-2020-05-10-00-46-23
convselfatt-2020-05-09-22-42-38
rnn-2020-05-09-22-32-36
selfatt-2020-05-08-19-04-10
selfatt-2020-04-24-20-26-59
selfatt-2020-04-24-18-40-59
400
300
200
100
0
Step
50
100
150
200
250

val-time-sec
convselfatt-2020-05-10-04-02-14
selfatt-2020-05-10-02-12-55
1dcnn-2020-05-10-01-14-09
selfatt-2020-05-10-00-46-45
neuralbow-2020-05-10-00-46-23
convselfatt-2020-05-09-22-42-38
rnn-2020-05-09-22-32-36
selfatt-2020-05-08-19-04-10
selfatt-2020-04-24-20-26-59
selfatt-2020-04-24-18-40-59
14
12
10
8
6
4
2
0
Step
50
100
150
200
250


val-loss
convselfatt-2020-05-10-04-02-14
selfatt-2020-05-10-02-12-55
1dcnn-2020-05-10-01-14-09
selfatt-2020-05-10-00-46-45
neuralbow-2020-05-10-00-46-23
convselfatt-2020-05-09-22-42-38
rnn-2020-05-09-22-32-36
selfatt-2020-05-08-19-04-10
selfatt-2020-04-24-20-26-59
selfatt-2020-04-24-18-40-59
Step
0
1
2
3
4
5
50
100
150
200
250


val-mrr
convselfatt-2020-05-10-04-02-14
selfatt-2020-05-10-02-12-55
1dcnn-2020-05-10-01-14-09
selfatt-2020-05-10-00-46-45
neuralbow-2020-05-10-00-46-23
convselfatt-2020-05-09-22-42-38
rnn-2020-05-09-22-32-36
selfatt-2020-05-08-19-04-10
selfatt-2020-04-24-20-26-59
selfatt-2020-04-24-18-40-59
Step
0
0.1
0.2
0.3
0.4
0.5
50
100
150
200
250

Discussion

From the experiments we can observe that the self-attention model got the first place, with CNN-self-attention hybrid model following as second, RNN third one, bag-of-word and CNN at last with similar performance.

It is not surprising there being a gap between self attention and traditional neural networks. Once again the self attention mechanism proved to be the currently state-of-art method in language tasks. Considering we are only using a minimal 3/128 transformer, we anticipate a larger transformer or even a standard one at size 12/768 will lead to an even better output.

RNN is the best among traditional neural networks which is neither beyond our expectation. Considering there are only limited keywords in the python programming language, the vocabulary is rather smaller than a human language. This fact leads to individual words carrying less information, and the relative positioning of words more important. Our experiment shows that RNN is better at capturing such relations, and despite being able to handle positional relations within its window, CNN does not outperform bag-of-words too much.

However, there are also abnormalities in the result. Although it is more complex, has more parameters and therefore looks promising, the CNN-self-attention hybrid model turned out to harm the performance eventually. A possible explanation can be that the parameters it adds to the model does not help the model to approximate the target function, but instead enlarges the hypothesis set for no obvious benefit. The extra CNN could have interfered with the self attention layers. This result showed that a more complex model may not always be better, and a careful design is required.

As for our improvement attempt, even though we lowered the depth of transformer from 3/128 to 2/128, we did not observe a noticeable difference after applying the pretrained parameters. As a side effect, we also lowered the running time from 1h49m to 1h22m which is only 75% of what it was before. Google's Bert-Tiny was trained on a far larger corpus (Wikipedia + BookCorpus), and our experiment illustrated that transferring this knowledge to our discussed domain is a viable way to improve performance. Unfortunately, we could only experiment with the minimal 2/128 BERT-Tiny due to limited resources, but we still consider this a possible way to push the performance further as future work.

Conclusion

We followed the idea from *Semantic Code Search*. In this project we focused only on Python language, and reproduced all 5 models referred on github:
Neural-Bag-Of-Words, RNN, 1D-CNN, Self-Attention (BERT), and a 1D-CNN+Self-Attention Hybrid.

To further improve the performance over the Elastic Search baseline, we developed our own model by loading the latest pretrained BERT-Tiny parameters into the transformer. Our experiments can be found on wandb and have been submitted to a community benchmark.

Thanks for the help from our professor, TAs, teammates and the paper.