CS 224N Default Final Project: Question Answering

Due date: Sunday, March 18th 2018 (Note this has changed!) at 11:59pm PST.
Late days: You are allowed to use 3 late days maximum for this assignment, so that we can complete grading on time. If you are working in a group, it requires one late day *per person* to push the deadline back by a day. See the grading page on the website for more information.

This assignment can be completed in groups of up to 3 people. We encourage groups to work together productively so that all students understand the submitted system well. We ask that you abide by the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work (except as acknowledged) is done by yourself and your team members only. Please review the Honor Code section (Section 9) of this document.

Please review any additional resources posted on the assignment page at `http://cs224n.stanford.edu/default_project/index.html`.

Contents

1 Overview **2**
- 1.1 The SQuAD Challenge . . . 2
- 1.2 This project . . . 3

2 Getting Started **4**
- 2.1 Code overview . . . 4
- 2.2 Setup . . . 5

3 The SQuAD Data **6**

4 Training your first baseline **6**
- 4.1 Baseline model description . . . 6
- 4.2 Train the baseline . . . 8
- 4.3 Tracking progress in TensorBoard . . . 8
- 4.4 Inspecting Output . . . 9

5 Improvements **10**
- 5.1 More complex attention . . . 10
  - 5.1.1 Bidirectional attention flow . . . 10
  - 5.1.2 Coattention . . . 11
  - 5.1.3 Self-attention . . . 12
  - 5.1.4 More attention techniques: . . . 12
- 5.2 Character-level CNN . . . 12
- 5.3 Conditioning End Prediction on Start Prediction . . . 13
- 5.4 Span Representations . . . 13
- 5.5 Additional input features . . . 14
- 5.6 Iterative reasoning (advanced) . . . 14
- 5.7 Other tips for improvement . . . 14

6 Alternative Goals **15**

7 Submitting to CodaLab **16**

8 Grading Criteria **16**

9 Honor Code **16**

10 FAQs **18**
- 10.1 How are out-of-vocabulary words handled? . . . 18
- 10.2 How are padding and truncation handled? . . . 18
- 10.3 Which parts of the code can I change? . . . 18

1 Overview

In this project, you will become a Deep Learning NLP researcher! You will develop a deep learning system for the Stanford Question Answering Dataset (SQuAD) [1].

1.1 The SQuAD Challenge

SQuAD is a *reading comprehension* dataset. This means your model will be given a paragraph, and a question about that paragraph, as input. The goal is to answer the question correctly. From a research perspective, this is an interesting task because it provides a measure for how well systems can 'understand' text. From a more practical perspective, this sort of question answering system could be extremely useful in the future. Imagine being able to ask an AI system questions so you can better understand any piece of text – like a class textbook, or a bill that is being signed into law.

SQuAD is less than two years old, but has already led to many research papers and significant breakthroughs in building effective reading comprehension systems. On the SQuAD webpage (https://rajpurkar.github.io/SQuAD-explorer/) there is a public leaderboard showing the performance of many systems. In January 2018, a team submitted a model that for the first time surpasses human performance according to one of the evaluation metrics!

The paragraphs in SQuAD are from Wikipedia. The questions and answers were crowdsourced using Amazon Mechanical Turk. There are around 100K questions in total. An important feature of SQuAD is that the answers are always taken directly from the paragraph. This means SQuAD systems don't have to generate the answer text – they just have to select the "span" of text in the paragraph that answers the question (imagine your model has a highlighter and needs to highlight the answer). Below is an example of a ⟨question, context, answer⟩ triple. To see more examples, you can explore the dataset on the website https://rajpurkar.github.io/SQuAD-explorer/explore/1.1/dev/.

> **Question**: Why was Tesla returned to Gospic?
> **Context paragraph**: On 24 March 1879, Tesla was returned to Gospic under police guard for not having a residence permit. On 17 April 1879, Milutin Tesla died at the age of 60 after contracting an unspecified illness (although some sources say that he died of a stroke). During that year, Tesla taught a large class of students in his old school, Higher Real Gymnasium, in Gospic.
> **Answer**: not having a residence permit

In fact, in the dev and test set, every SQuAD question has *three answers* provided – each answer from a different crowd worker. The answers don't always completely agree, which is partly why 'human performance' on the SQuAD leaderboard is not 100%. Performance is measured via two metrics: **F1** and **Exact Match (EM)** score.

- Exact Match is a binary measure (i.e. true/false) of whether the system output matches the ground truth answer exactly. For example, if your system answered a question with 'Einstein' but the ground truth answer was 'Albert Einstein', then you would get an EM score of 0 for that example. This is a fairly strict metric!
- F1 is a less strict metric – it is the harmonic mean of precision and recall[1]. In the 'Einstein' example, the system would have 100% precision (its answer is a subset of the ground truth answer) and 50% recall (it only included one out of the two words in the ground truth output), thus a F1 score of $2 \times \text{prediction} \times \text{recall}/(\text{precision}+\text{recall}) = 2*50*100/(100+50) = 66.67\%$.

Given that the SQuAD dataset has *three* human-provided answers for each question, official evaluation takes the *maximum* F1 and EM scores across the three human-provided answers. This makes evaluation more forgiving – for example, if one of the human annotators *did* answer 'Einstein', then your system will get 100% EM and 100% F1 for that example.

Finally, the EM and F1 scores are averaged across the entire evaluation dataset to get the final reported scores.

[1] Read more about F1 here: https://en.wikipedia.org/wiki/F1_score

1.2 This project

The goal of this project is to produce a reading comprehension system that works well on SQuAD. We have provided code for preprocessing the data and computing the evaluation metrics, and code to train a fully-functional neural baseline. Your job is to improve on this baseline.

In Section 5, we describe several improvements that are commonly used in high-performing SQuAD models – most come from recent research papers. We provide these suggestions to help you get started improving your model. They should all improve over the baseline if implemented correctly (and note that there is usually more than one way to implement something correctly). Some of the methods are described quite thoroughly in this document. However, we don't describe every detail, so you should refer to the original papers to figure out all the details. In rare cases a detail won't even be in the paper, and you will need to use your own judgement to decide what to do.

We describe other improvements in less detail, although we do point to the relevant papers. These improvements are worth a bit more in terms of grading because they require more investigation on your part. (Of course, we also will take into account the complexity of the improvement and its effectiveness when we grade). You can even try doing some research and implement something original! This doesn't necessarily have to be a completely new approach – small but well-motivated changes to existing models are very valuable, especially if followed by good analysis. If you can show quantiatively and qualitatively that your small but original change improves a state-of-the-art model (and even better, explain what particular problem it solves and how), then you will have done extremely well.

In many cases there won't be one correct answer for how to do something – it will take experimentation to determine which way is best. We are expecting you to exercise the judgment and intuition that you've gained from the class so far to build your models. Note that there is *no minimum F1/EM score*, and *no minimum number of improvements* you should implement. Instead, you will be graded *holistically* based on what you implement, how well it works, and how clearly and thoroughly you analyze and describe your methods in the writeup – see Section 8 for more details.

2 Getting Started

For this project, you will need a machine with GPUs to train your models efficiently. For this, we encourage you to use Azure – please see the *Azure Guide* on the project page for details on how to set up your Virtual Machine (VM). However, you only have a finite amount of Azure credit, which is charged for every minute that your VM is on (see the Azure Guide for more information). Therefore, it is important that your VM is only turned on when you are actually training your models.

For this reason, we advise that you **develop your code on your local machine** (or one of the Stanford machines, like `rice`), with the CPU version of TensorFlow installed, and move to your Azure VM only once you've debugged your code and you're ready to train. We advise that you use Github to manage your codebase and sync it between the two machines – for more information on this, see the see the *Practical tips for final projects* document on the project page. Note: If you use Github to manage your code, you must keep your repository **private**.

When you work through this *Get Started* guide for the first time, do so on your local machine. You will then repeat the process on your Azure VM.

Once you are on an appropriate machine, clone the project Github repository with the following command.

```
git clone https://github.com/abisee/cs224n-win18-squad.git
```

We encourage you to `git clone` our repository, rather than simply download it, so that you can easily integrate any bug fixes that we make to the code. In fact, you should periodically check whether there are any new fixes that you need to download. To do so, navigate to the `cs224n-win18-squad` directory and run the `git pull` command.

2.1 Code overview

The repository `cs224n-win18-squad` contains the following files:

- `get_started.sh`: A script to install requirements, and download and preprocess the data.
- `requirements.txt`: Used by `get_started.sh` to install requirements.
- `docker/`: A directory containing a Dockerfile:
  - `Dockerfile`: This is the specification for the Docker image `abisee/cs224n-dfp:v4` which is available on Docker Hub at https://hub.docker.com/r/abisee/cs224n-dfp/. We provide this Dockerfile in case you need to build your own Docker image for submission to CodaLab – for more information see the CodaLab submission instructions on the project page.
- `code/`: A directory containing all code:
  - `preprocessing/`: Code to preprocess the SQuAD data, so it is ready for training:
    - `download_wordvecs.py`: Downloads and stores the pretrained word vectors (GloVe).
    - `squad_preprocess.py`: Downloads and preprocesses the official SQuAD train and dev sets and writes the preprocessed versions to file.
  - `data_batcher.py`: Reads the pre-processed data from file and processes it into batches for training.
  - `evaluate.py`: The official evaluation script from SQuAD. Your model can import evaluation functions from this file, but you should not change this file.
  - `main.py`: The top-level entrypoint to the code. You can run this file to train the model, view examples from the model and evaluate the model.
  - `modules.py`: Contains some basic model components, like a RNN Encoder module, a basic dot-product attention module, and a simple softmax layer. You will add modules to this file.

- `official_eval_helper.py`: Contains code to read an official SQuAD JSON file, use your model to predict answers, and write those answers to another JSON file. This is required for official evaluation. (See Section 7).
- `pretty_print.py`: Contains code to visualize model output.
- `qa_model.py`: Contains the model definition. You will do much of your work in this file, and in `modules.py`.
- `vocab.py`: Contains code to read GloVe embeddings from file and make them into an embedding matrix.

2.2 Setup

Once you are on an appropriate machine and have cloned the project repository, it's time to run the setup script.

- If you are on your **local machine**, first open `requirements.txt` and change the line `tensorflow-gpu==1.4.1` to `tensorflow==1.4.1`. This will ensure that the setup script installs the CPU version of TensorFlow on your machine.
- If you are on your **GPU machine** (e.g. your Azure VM), do not change `requirements.txt` (it should say `tensorflow-gpu==1.4.1`).

Now run the following commands:

```
cd cs224n-win18-squad     # Ensure we're in the project directory
./get_started.sh          # Run the startup script
```

Say yes when the script asks you to confirm that new packages will be installed. This script creates a new conda[2] environment for you called `squad`. You need to remember to activate your `squad` environment everytime you use the code.[3] To activate the `squad` environment, run

```
source activate squad
```

The `get_started.sh` script will also install dependencies, download data, and create new data files. It may take several minutes to complete. In particular, the script downloads 862MB of GloVe word vectors, which may take some time. Once the script has finished, you should now see the following additional files in `cs224n-win18-squad`:

- `data/`: A directory to contain all data.
  - `dev-v1.1.json`: The official SQuAD dev set.
  - `train-v1.1.json`: The official SQuAD train set.
  - `dev.{answer/context/question/span}`: Tokenized dev set data. All files have 10391 lines, each corresponding to a different example in the dev set.
  - `train.{answer/context/question/span}`: Tokenized train set data. All files have 86326 lines[4], each corresponding to a different example in the train set.
  - `glove.6B.zip`: Zipped GloVe vectors. These were trained on 6 billion words of Wikipedia and Gigaword. See more information here: https://nlp.stanford.edu/projects/glove/.
  - `glove.6B.{50/100/200/300}d.txt`: GloVe vectors of dimensionality 50/100/200/300, respectively. Each file has 400k lines, corresponding to 400k unique lowercase words.
- `experiments/`: A (currently empty) directory to store data from experiments.

If you see all of these files, then you're ready to get started training the baseline model (see Section 4.2)! If not, check the output of `get_started.sh` for error messages, and ask for assistance on Piazza if applicable.

[2] Conda is a package and environment management system. You can learn the basics of Conda here: https://conda.io/docs/user-guide/getting-started.html

[3] You can tell that the `squad` environment is activated because your command prompt says `(squad)`.

[4] We've found that on some systems (like OSX), this is sometimes 86318 due to Unicode issues. This is fine. See note in `squad_preprocess.py` for explanation.

3 The SQuAD Data

The SQuAD dataset has three splits: **train**, **dev** and **test**. The train and dev sets are publicly available, while the test set is kept secret. The official leaderboard[5] shows performance on the test set.

You will use the train set to train your model and the dev set to tune hyperparameters. Finally, you will submit your models to a class leaderboard hosted by CodaLab (see `http://codalab.org`) where they will be evaluated on both the dev set and the test set (see Section 7).

The SQuAD dataset contains many (context, question, answer) triples[6] – see an example in Section 1.1. Each *context* (sometimes called a *passage*, *paragraph* or *document* in other papers) is an excerpt from Wikipedia. The *question* (sometimes called a *query* in other papers) is the question to be answered based on the context. The *answer* is a span (i.e. excerpt of text) from the context.

In the `data` directory, you should find the preprocessed versions of the training and dev data. All files have one example per line.

- `{train/dev}.answer`: The answer text, tokenized and lowercased.
- `{train/dev}.context`: The context, tokenized and lowercased.
- `{train/dev}.question`: The question, tokenized and lowercased.
- `{train/dev}.span`: The indices of the first and last tokens of the answer (inclusive). The indices are given with respect to the context tokens (zero-indexed).[7]

> **Suggestion**: Write a script (perhaps in a Jupyter notebook) to produce a histogram plot of the lengths (in tokens) of the context, question, and answer in the training data. Collect statistics about where in the context the answer appears (e.g. is the answer more likely to appear at the beginning or the end of the context?). These statistics will be important when you are deciding, for example, what the cut-off should be for whether you truncate and/or discard contexts and questions that are too long (these hyperparameters are called `context_len` and `question_len` in the code). It is *always* worth getting to know your data before starting work on building a solution, and it will help you get started on the analysis section of your writeup later.

4 Training your first baseline

We have provided you with the complete code for a simple baseline model, which uses deep learning techniques you learned in class. In this section we will describe the baseline model and show you how to train it.

4.1 Baseline model description

Our baseline model has three components: **a RNN encoder layer**, that encodes both the context and the question into hidden states, **an attention layer**, that combines the context and question representations, and an **output layer**, which applies a fully connected layer and then two separate softmax layers (one to get the start location, and one to get the end location of the answer span). All these modules can be found in `modules.py`, and the code to connect them is in `qa_model.py`.

RNN Encoder Layer: For each SQuAD example (context, question, answer), the context is represented by a sequence of d-dimensional word embeddings $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N \in \mathbb{R}^d$, and the question by a sequence of d-dimensional word embeddings $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_M \in \mathbb{R}^d$. These are fixed, pre-trained

[5] `https://rajpurkar.github.io/SQuAD-explorer/`

[6] As described in Section 1.1, the dev and test sets actually have *three* human-provided answers for each question. But the training set only has one answer per question.

[7] The original SQuAD data gives the answer spans w.r.t. character counts. In `squad_preprocess.py` we convert these to spans w.r.t. token counts. This means that we have to lose some examples from the dataset, due to tokenization special cases – see `squad_preprocess.py` for more information.

GloVe embeddings. The embeddings are fed into a 1-layer bidirectional GRU (which is shared between the context and the question):

$$\{\overrightarrow{\boldsymbol{c}_1}, \overleftarrow{\boldsymbol{c}_1}, \ldots, \overrightarrow{\boldsymbol{c}_N}, \overleftarrow{\boldsymbol{c}_N}\} = \text{biGRU}(\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\})$$
$$\{\overrightarrow{\boldsymbol{q}_1}, \overleftarrow{\boldsymbol{q}_1}, \ldots, \overrightarrow{\boldsymbol{q}_M}, \overleftarrow{\boldsymbol{q}_M}\} = \text{biGRU}(\{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_M\})$$

The bidirectional GRU produces a sequence of forward hidden states ($\overrightarrow{\boldsymbol{c}_i} \in \mathbb{R}^h$ for the context and $\overrightarrow{\boldsymbol{q}_j} \in \mathbb{R}^h$ for the question) and a sequence of backward hidden states ($\overleftarrow{\boldsymbol{c}_i}$ and $\overleftarrow{\boldsymbol{q}_j}$). We concatenate the forward and backward hidden states to obtain the *context hidden states* $\boldsymbol{c}_i$ and the *question hidden states* $\boldsymbol{q}_j$ respectively:

$$\boldsymbol{c}_i = [\overrightarrow{\boldsymbol{c}_i}; \overleftarrow{\boldsymbol{c}_i}] \in \mathbb{R}^{2h} \quad \forall i \in \{1, \ldots, N\}$$
$$\boldsymbol{q}_j = [\overrightarrow{\boldsymbol{q}_j}; \overleftarrow{\boldsymbol{q}_j}] \in \mathbb{R}^{2h} \quad \forall j \in \{1, \ldots, M\}$$

Attention Layer: Next, we apply basic dot-product attention, with the context hidden states $\boldsymbol{c}_i$ attending to the question hidden states $\boldsymbol{q}_j$. For each context hidden state $\boldsymbol{c}_i$, the *attention distribution* $\alpha^i \in \mathbb{R}^M$ is computed as follows:

$$\boldsymbol{e}^i = [\boldsymbol{c}_i^T \boldsymbol{q}_1, \ldots, \boldsymbol{c}_i^T \boldsymbol{q}_M] \in \mathbb{R}^M$$
$$\alpha^i = \text{softmax}(\boldsymbol{e}^i) \in \mathbb{R}^M$$

The attention distribution is then used to take a weighted sum of the question hidden states $\boldsymbol{q}_j$, producing the *attention output* $\boldsymbol{a}_i$:

$$\boldsymbol{a}_i = \sum_{j=1}^{M} \alpha_j^i \boldsymbol{q}_j \in \mathbb{R}^{2h}$$

The attention outputs are then concatenated to the context hidden states to obtain the *blended representations* $\boldsymbol{b}_i$:

$$\boldsymbol{b}_i = [\boldsymbol{c}_i; \boldsymbol{a}_i] \in \mathbb{R}^{4h} \quad \forall i \in \{1, \ldots, N\}$$

Output Layer: Next, each of the blended representations $\boldsymbol{b}_i$ are fed through a fully connected layer followed by a ReLU non-linearity:

$$\boldsymbol{b}'_i = \text{ReLU}(\boldsymbol{W}_{FC} \boldsymbol{b}_i + \boldsymbol{v}_{FC}) \in \mathbb{R}^h \quad \forall i \in \{1, \ldots, N\}$$

where $\boldsymbol{W}_{FC} \in \mathbb{R}^{h \times 4h}$ and $\boldsymbol{v}_{FC} \in \mathbb{R}^h$ are a weight matrix and bias vector. Next, we assign a score (or logit) to each context location i by passing $\boldsymbol{b}'_i$ through a downprojecting linear layer:

$$\text{logits}_i^{\text{start}} = \boldsymbol{w}_{\text{start}}^T \boldsymbol{b}'_i + u_{\text{start}} \in \mathbb{R} \quad \forall i \in \{1, \ldots, N\}$$

where $\boldsymbol{w}_{\text{start}} \in \mathbb{R}^h$ is a weight vector and $u_{\text{start}} \in \mathbb{R}$ a bias term. Finally, we apply the softmax function to $\text{logits}_{\text{start}} \in \mathbb{R}^N$ to obtain a probability distribution $p^{\text{start}} \in \mathbb{R}^N$ over the context locations $\{1, \ldots, N\}$:

$$p^{\text{start}} = \text{softmax}(\text{logits}^{\text{start}}) \in \mathbb{R}^N$$

We compute a probability distribution p^{end} in the same way (though with separate weights $\boldsymbol{w}_{\text{end}}$ and u_{end}).

Loss: Our loss function is the sum of the cross-entropy loss for the start and end locations. That is, if the gold start and end locations are $i_{\text{start}} \in \{1, \ldots, N\}$ and $i_{\text{end}} \in \{1, \ldots, N\}$ respectively, then the loss for a single example is:

$$\text{loss} = -\log p^{\text{start}}(i_{\text{start}}) - \log p^{\text{end}}(i_{\text{end}})$$

During training, this loss is averaged across the batch and minimized with the Adam optimizer.

Prediction: At test time, given a context and a question, we simply take the argmax over p^{start} and p^{end} to obtain the predicted span $(\ell^{\text{start}}, \ell^{\text{end}})$:[8]

$$\ell^{\text{start}} = \text{argmax}_{i=1}^{N} p_i^{\text{start}}$$
$$\ell^{\text{end}} = \text{argmax}_{i=1}^{N} p_i^{\text{end}}$$

[8] If $\ell^{\text{start}} > \ell^{\text{end}}$, then the predicted answer text is just an empty string.

4.2 Train the baseline

Before starting to train the baseline, consider opening a new session with `tmux` or some other session manager. This will make it easier for you to leave your model training for a long time, then retrieve the session later. For more information about TMUX, see the *Practical tips for final projects* document on the projects page.

To start training the baseline, run the following commands:

```
source activate squad   # Remember to always activate your squad environment
cd code                 # Change to code directory
python main.py --experiment_name=baseline --mode=train  # Start training
```

After some initialization, you should see the model begin to log information like the following:

```
epoch 1, iter 41, loss 8.29449, smoothed loss 9.33690, grad norm 1.52926, \
  param norm 48.66283, batch time 1.274
```

You should see the loss begin to drop. If you're training on an Azure NV6 machine (which has one GPU), you should see batches taking around 1.1 to 1.3 seconds on average. On a Macbook Pro laptop running CPU-only Tensorflow, batches take around 4.7 to 5.0 seconds on average.

You should also see that there is a new subdirectory of `experiments` called `baseline`. This is where you can find all data relating to this experiment. In particular, you will (eventually) see:

- `flags.json`: A record of the flags that you passed into `main.py` when you began the experiment.
- `log.txt`: A record of all the logging during training.
- `events.out.tfevents.*`: These files contain information (like the loss over time) for TensorBoard to visualize.
- `qa.ckpt-xxx.*`: These are checkpoint files, that contain the weights of the latest version of the model. The number `xxx` corresponds to how many training iterations had been completed when the model was saved. By default the first checkpoint is saved after the first 500 iterations, but you can save checkpoints more frequently by changing the `save_every` flag.
- `checkpoint`: A text file containing a pointer to the latest `qa.ckpt-xxx` checkpoint files.
- `best_checkpoint/`: A directory containing the *best* model so far (i.e. the model achieving highest dev set EM score). These checkpoints are saved with the name `qa_best.ckpt-xxx`.

4.3 Tracking progress in TensorBoard

We strongly encourage you to use TensorBoard – it's one of the best features of TensorFlow and will enable you to get a much better view of your experiments. To use TensorBoard, run the following command:

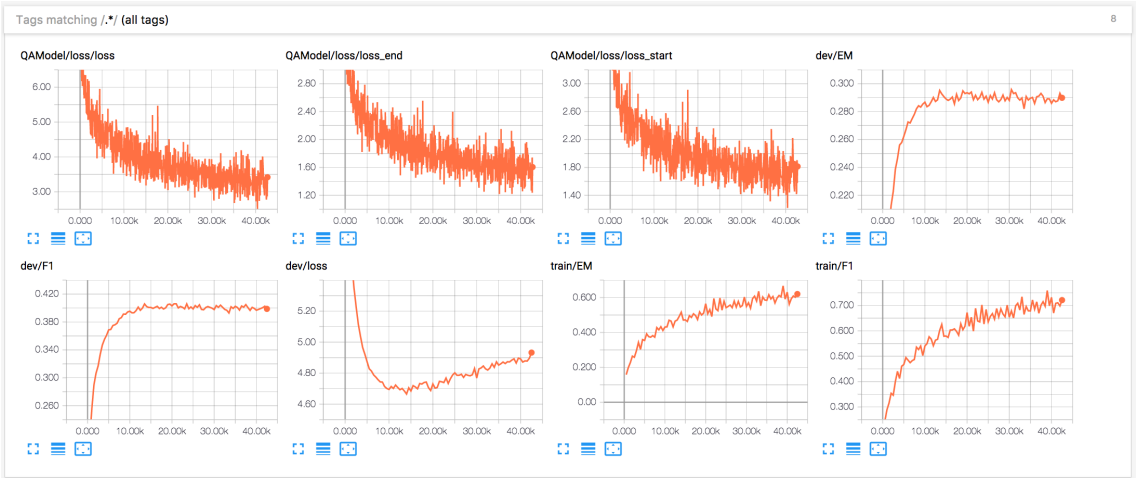
```
cd experiments                        # Go to experiments directory
tensorboard --logdir=. --port=5678    # Start TensorBoard
```

- If you are training on your local machine, now open http://localhost:5678/ in your browser.
- If you are training on a remote machine (e.g. Azure), then run the following command on your local machine:

  ```
  ssh -N -f -L localhost:1234:localhost:5678 <user>@<remote>
  ```

  where `<user>@<remote>` is the address that you `ssh` to for your remote machine. Then on your local machine, open http://localhost:1234/ in your browser.

You should see TensorBoard load, with plots of the loss, EM and F1 for both train and dev sets. The F1 and EM plots may take some time to appear because they are logged less frequently than the loss. However, you should see train loss decreasing from the very start. Here is the view after 40k iterations:



In particular, over 40k iterations we find that:

- the train loss, train F1 and train EM scores continue to improve throughout
- the dev loss begins to rise around 10-15k iterations (overfitting)
- the dev F1 and EM scores reach around 39 and 28 respectively, then plateau.

If we assume that the model begins overfitting and therefore reaches its peak performance at 15k iterations, then on an Azure NV6 machine, it will take you approximately 15k iterations = 5 hours to achieve peak performance.

We advise you to reproduce this experiment (i.e. train the baseline and obtain results similar to those we report above) – however, you can stop training earlier (i.e. around 15k iterations rather than 40k). This will give you something to compare your improved models against. In particular, TensorBoard will plot your new experiments overlaid with your baseline experiment – this will enable you to see how your improved models progress over time, compared to the baseline.

4.4 Inspecting Output

Once you have a trained model, you will want to see example output to help you begin to think about error analysis, and how you might improve the model. Run the following command.

```
source activate squad   # Remember to always activate your squad environment
python main.py --experiment_name=baseline --mode=show_examples
```

You should see ten random dev set examples printed to screen, comparing the true answer to your model's predicted answer, and giving the F1 and EM score for each example. This should help you start to notice the main weaknesses of the model, and think about how you could fix them. After measuring statistics of the data, training the baseline, looking at TensorBoard and inspecting output, you should have some intuition about about how the baseline can be improved!

5 Improvements

From here, the project is open-ended! Your job is to investigate various models for SQuAD, and build the best system you can. As explained in Section 1.2, in this section we are providing you with an overview of common techniques that are used in high-performing SQuAD models. Your job will be to choose some of these improvements, understand them, implement them, carefully train them, and analyze their performance. Understanding these improvements will frequently involve reading the original research paper, or other materials. Implementing them is an open-ended task: for each improvement there are multiple valid implementations. To learn more about how these improvements will be graded, and how much you are expected to do, see Section 8.

Note: The notation used in this section will sometimes differ from that used in the original research papers – this is so that we can use consistent notation within this document.

5.1 More complex attention

As we learned in lectures, there are many types of attention, some of which are more complex than the basic dot-product attention used in our baseline. Attention is a key component of almost all high-performing SQuAD models, and it's typically used to combine the representations for the context and the question.

5.1.1 Bidirectional attention flow

Appears in: BiDAF [2].

BiDAF is a high-performing SQuAD model. The core part of the model is the Bidirectional Attention Flow layer, which we will describe here. The main idea is that attention should flow both ways – from the context to the question and from the question to the context. The Bidirectional Attention Flow layer could be substituted into the baseline in place of the basic Attention Layer.

Assume we have context hidden states $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N \in \mathbb{R}^{2h}$ and question hidden states $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_M \in \mathbb{R}^{2h}$. We compute the *similarity matrix* $\boldsymbol{S} \in \mathbb{R}^{N \times M}$, which contains a similarity score $\boldsymbol{S}_{ij}$ for each pair $(\boldsymbol{c}_i, \boldsymbol{q}_j)$ of context and question hidden states.

$$\boldsymbol{S}_{ij} = \boldsymbol{w}_{\text{sim}}^T [\boldsymbol{c}_i; \boldsymbol{q}_j; \boldsymbol{c}_i \circ \boldsymbol{q}_j] \in \mathbb{R}$$

Here, $\boldsymbol{c}_i \circ \boldsymbol{q}_j$ is an elementwise product and $\boldsymbol{w}_{\text{sim}} \in \mathbb{R}^{6h}$ is a weight vector.

Next, we perform Context-to-Question (C2Q) Attention. (This is similar to our baseline's Attention Layer). We take the row-wise softmax of $\boldsymbol{S}$ to obtain attention distributions α^i, which we use to take weighted sums of the question hidden states $\boldsymbol{q}_j$, yielding *C2Q attention outputs* $\boldsymbol{a}_i$. In equations, this is:

$$\begin{aligned} \alpha^i &= \text{softmax}(\boldsymbol{S}_{i,:}) \in \mathbb{R}^M \quad \forall i \in \{1, \ldots, N\} \\ \boldsymbol{a}_i &= \sum_{j=1}^{M} \alpha_j^i \boldsymbol{q}_j \in \mathbb{R}^{2h} \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

Next, we perform Question-to-Context(Q2C) Attention. For each context location $i \in \{1, \ldots, N\}$, we take the max of the corresponding row of the similarity matrix, $\boldsymbol{m}_i = \max_j \boldsymbol{S}_{ij} \in \mathbb{R}$. Then we take the softmax over the resulting vector $\boldsymbol{m} \in \mathbb{R}^N$ – this gives us an attention distribution $\beta \in \mathbb{R}^N$ over context locations. We then use β to take a weighted sum of the context hidden states $\boldsymbol{c}_i$ – this is the *Q2C attention output* $\boldsymbol{c}'$. In equations:

$$\begin{aligned} \boldsymbol{m}_i &= \max_j \boldsymbol{S}_{ij} \in \mathbb{R} \quad \forall i \in \{1, \ldots, N\} \\ \beta &= \text{softmax}(\boldsymbol{m}) \in \mathbb{R}^N \\ \boldsymbol{c}' &= \sum_{i=1}^{N} \beta_i \boldsymbol{c}_i \in \mathbb{R}^{2h} \end{aligned} \tag{1}$$

Lastly, for each context location $i \in \{1, \ldots, N\}$ we obtain the output $\boldsymbol{b}_i$ of the Bidirectional Attention Flow Layer by combining the context hidden state $\boldsymbol{c}_i$, the C2Q attention output $\boldsymbol{a}_i$, and

the Q2C attention output $\boldsymbol{c}'$:

$$\boldsymbol{b}_i = [\boldsymbol{c}_i; \boldsymbol{a}_i; \boldsymbol{c}_i \circ \boldsymbol{a}_i; \boldsymbol{c}_i \circ \boldsymbol{c}'] \in \mathbb{R}^{8h} \quad \forall i \in \{1, \dots, N\}$$

where $\circ$ represents elementwise multiplication.

5.1.2 Coattention

Appears in: Dynamic Coattention Network [3].

The Dynamic Coattention Network is another high-performing SQuAD model. One of its two main contributions is the Coattention Layer, which, like BiDAF, involves a two-way attention between the context and the question. However, unlike Bidirectional Attention Flow, Coattention involves a *second-level attention computation* – i.e., attending over representations that are themselves attention outputs. Here, we describe the Coattention Layer, which could be substituted into the baseline in place of the basic Attention Layer.

Assume we have context hidden states $\boldsymbol{c}_1, \dots, \boldsymbol{c}_N \in \mathbb{R}^l$ and question hidden states $\boldsymbol{q}_1, \dots, \boldsymbol{q}_M \in \mathbb{R}^l$. First, apply a linear layer with tanh nonlinearity to the question hidden states to obtain projected question hidden states $\boldsymbol{q}'_1, \dots, \boldsymbol{q}'_M$:

$$\boldsymbol{q}'_j = \tanh(\boldsymbol{W}\boldsymbol{q}_j + \boldsymbol{b}) \in \mathbb{R}^l \quad \forall j \in \{1, \dots, M\}$$

where $\boldsymbol{W}$ is a weight matrix and $\boldsymbol{b}$ a bias vector. Next, add *sentinel vectors* $\boldsymbol{c}_\emptyset \in \mathbb{R}^l$ and $\boldsymbol{q}'_\emptyset \in \mathbb{R}^l$ (which are a trainable parameters of the model) to both the context and the question states. This gives us $\{\boldsymbol{c}_1, \dots, \boldsymbol{c}_N, \boldsymbol{c}_\emptyset\}$ and $\{\boldsymbol{q}'_1, \dots, \boldsymbol{q}'_M, \boldsymbol{q}'_\emptyset\}$.[9]

Next, we compute the *affinity matrix* $\boldsymbol{L} \in \mathbb{R}^{(N+1)\times(M+1)}$, which contains the affinity score $\boldsymbol{L}_{ij}$ for each pair $(\boldsymbol{c}_i, \boldsymbol{q}'_j)$ of context and question hidden states:

$$\boldsymbol{L}_{ij} = \boldsymbol{c}_i^T \boldsymbol{q}'_j \in \mathbb{R}$$

Next, we use the affinity matrix $\boldsymbol{L}$ to compute attention outputs for both directions.
For the Context-to-Question (C2Q) Attention, we obtain *C2Q attention outputs* $\boldsymbol{a}_i$:

$$\alpha^i = \text{softmax}(\boldsymbol{L}_{i,:}) \in \mathbb{R}^{M+1}$$
$$\boldsymbol{a}_i = \sum_{j=1}^{M+1} \alpha^i_j \boldsymbol{q}'_j \in \mathbb{R}^l$$

For the Question-to-Context (Q2C) Attention, we obtain *Q2C attention outputs* $\boldsymbol{b}_j$:

$$\beta^j = \text{softmax}(\boldsymbol{L}_{:,j}) \in \mathbb{R}^{N+1}$$
$$\boldsymbol{b}_j = \sum_{i=1}^{N+1} \beta^j_i \boldsymbol{c}_i \in \mathbb{R}^l$$

Next, we use the C2Q attention distributions α^i to take weighted sums of the Q2C attention outputs $\boldsymbol{b}_j$. This gives us *second-level attention outputs* $\boldsymbol{s}_i$:

$$\boldsymbol{s}_i = \sum_{j=1}^{M+1} \alpha^i_j \boldsymbol{b}_j \in \mathbb{R}^l \quad \forall i \in \{1, \dots, N\}$$

Finally, we concatenate the second-level attention outputs $\boldsymbol{s}_i$ with the first-level C2Q attention outputs $\boldsymbol{a}_i$, and feed the sequence through a bidirectional LSTM. The resulting hidden states $\boldsymbol{u}_i$ of this biLSTM are known as the *coattention encoding*. This is the overall output of the Coattention Layer.

$$\{\boldsymbol{u}_1, \dots, \boldsymbol{u}_N\} = \text{biLSTM}(\{[\boldsymbol{s}_1; \boldsymbol{a}_1], \dots, [\boldsymbol{s}_N; \boldsymbol{a}_N]\})$$

[9] The purpose of the sentinel vectors is to make it possible to attend to none of the provided hidden states. See the paper for more details.

5.1.3 Self-attention

Appears in: R-Net[10]

R-Net is a simple but high-performing SQuAD model that has both a Context-to-Question attention layer (similar to our baseline), and a self-attention layer (which they call Self-Matching Attention). Both layers are simple applications of additive attention (as described in Lecture 11).

The self-attention layer is as follows. Suppose we have some sequence of representations $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N \in \mathbb{R}^\ell$, with each $\boldsymbol{v}_i$ corresponding to a context location $i \in \{1, \ldots, N\}$. Each $\boldsymbol{v}_i$ attends to *all* the $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N\}$.[11] In equations:

$$\begin{aligned} \boldsymbol{e}_j^i &= \mathbf{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{v}_j + \boldsymbol{W}_2 \boldsymbol{v}_i) \in \mathbb{R} \\ \alpha^i &= \text{softmax}(\boldsymbol{e}^i) \in \mathbb{R}^N \\ \boldsymbol{a}^i &= \sum_{j=1}^{N} \alpha_j^i \boldsymbol{v}_j \in \mathbb{R}^\ell \end{aligned}$$

Here $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are weight matrices and $\mathbf{v}$ is a weight vector. We then concatenate the self-attention output $\boldsymbol{a}_i$ to $\boldsymbol{v}_i$ for each i, and pass these as input to a bidirectional RNN to obtain a new set of hidden states $\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N\}$:

$$\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N\} = \text{biRNN}(\{[\boldsymbol{v}_1; \boldsymbol{a}_1], \ldots, [\boldsymbol{v}_N; \boldsymbol{a}_N]\})$$

The representations $\{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N\}$ are the output of the self-attention layer.

You could insert this self-attention layer into the baseline model, for example after the basic Attention Layer. However, to replicate R-Net, there are more things you'd need to change, so read the paper. (It's one of the easier papers to understand!)

5.1.4 More attention techniques:

- Fine-Grained Gating [4].
- Multi-Perspective Matching [5].

5.2 Character-level CNN

Appears in: BiDAF [2].

As we learned in lectures, character-level encodings are becoming increasingly popular recently. Compared to word vectors, they offer the advantages of allowing us to condition on the internal structure of words (this is known as *morphology*), and better handle out-of-vocabulary words.

Suppose we have a word w which consists of characters $c_1, \ldots, c_L$. We usually start by representing the characters using trainable *character embeddings* $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_L$. You can think of character embeddings as analogous to word embeddings, though character embeddings typically have smaller dimension d_c, and the 'vocabulary size' is much smaller (just the size of the alphabet, plus some digits and punctuation).

In a character-level Convolutional Neural Network (CNN), we take our character embeddings $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_L \in \mathbb{R}^{d_c}$, and compute another sequence of hidden representations $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_L \in \mathbb{R}^f$. The core idea is that each $\boldsymbol{h}_i$ is computed based on a *window* of characters $[c_{i-k}, \ldots, c_i, \ldots, c_{i+k}]$ centered at position i (k is the window width). Finally, you apply elementwise *max pooling* to obtain the character-level encoding: $\text{emb}_{\text{char}}(w) = \max_i \boldsymbol{h}_i \in \mathbb{R}^f$.

Typically, once we have obtained the character-level encoding $\text{emb}_{\text{char}}(w)$, we concatenate it with the usual pretrained word embedding $\text{emb}_{\text{word}}(w)$ to get a hybrid representation for w. You could augment the baseline by using hybrid representations in place of just pretrained word embeddings. If you choose to implement this, you will need to start by creating a new `tf.placeholder`

[10] https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/r-net.pdf

[11] Note this is different to the self-attention for Language Models described in Lecture 11, where each hidden state attends only to the *previous* hidden states. Here, we can attend to *all* other context locations because we have access to the entire context. By contrast, in a Language Model setting we do not have access to the future words.

of shape `(batch_size, context_len, word_len)` to feed in the character IDs for e.g. the context. (`word_len`, similar to `context_len`, will be the maximum word length that the model can process).

To get started with character CNNs, look at the TensorFlow function `tf.layers.conv1d`. The two important parameters are `filters`, which corresponds to f (dimension of the output states $\boldsymbol{h}_i$) here, and `kernel_size`, which corresponds to k (window width) here. We found that $d_c = 20, k = 5, f = 100$ gives reasonable performance, though we did not perform an exhaustive hyperparameter search. You could also construct a multi-layer character-CNN (perhaps with different f and k on each layer). You can learn more about CNNs for NLP here[12].

5.3 Conditioning End Prediction on Start Prediction

Appears in: Match-LSTM and Answer Pointer [6].

Note: See Piazza post about this model.

Our baseline predicts the start location and the end location *independently*, given the final layer's activations $\boldsymbol{b}'$. Instead, you could build a model that conditions the probability distribution for the end location on the start location probability distribution.

The Answer Pointer component of the 'Match-LSTM with Answer Pointer' model[13] does this. The Answer Pointer is similar to the decoder component of a sequence-to-sequence model, like we saw in lectures. However, it is a Pointer Network, meaning that on each timestep, instead of outputting a probability distribution over a vocabulary (like a NMT decoder), it outputs a probability distribution over locations in the context (i.e. locations it could *point to*).

Suppose we have a set of representations $\boldsymbol{h}_1^r, \ldots, \boldsymbol{h}_N^r \in \mathbb{R}^\ell$, one for each context location. The Answer Pointer is a RNN that is run for exactly two timesteps. On the first timestep, the Answer Pointer hidden state attends to $\boldsymbol{h}_1^r, \ldots, \boldsymbol{h}_N^r$, producing an attention distribution $\beta_s \in \mathbb{R}^N$ and an attention output $\boldsymbol{a}_s \in \mathbb{R}^\ell$. We use β_s as p^{start}, the probability distribution for the start location. On the second timestep, the attention output $\boldsymbol{a}_s$ is used as input to the Answer Pointer RNN (this is how p^{end} depends on p^{start}). Then we use the new Answer Pointer hidden state to attend to $\boldsymbol{h}_1^r, \ldots, \boldsymbol{h}_N^r$, producing an attention distribution $\beta_e \in \mathbb{R}^N$ which we use as p^{end}.

You could implement a system like this to replace the Output Layer of our baseline. You could additionally read the Match-LSTM and Answer Pointer paper to learn about the other techniques used in the model (for example, the Match-LSTM component).

5.4 Span Representations

Appears in: Dynamic Chunk Reader [7].

The SQuAD task can be phrased as the task of determining the joint probability distribution

$$P(\ell^{\text{start}}, \ell^{\text{end}} | \text{context}, \text{question})$$

where ℓ^{start} and ℓ^{end} are random variables corresponding to the start and end locations of the true answer span. The majority of SQuAD models (including our baseline) calculate this as the product of the probabilities of the start and end positions:

$$P(\ell^{\text{start}}, \ell^{\text{end}} | \text{context}, \text{question}) = P(\ell^{\text{start}} | \text{context}, \text{question}) P(\ell^{\text{end}} | \text{context}, \text{question}).$$

As an alternative approach, you could build a model that *directly* computes the probability of each possible *span*. This is what the Dynamic Chunk Reader (DCR) does. The DCR model first obtains a list of candidate spans. This can be done, for example, by enumerating all possible spans of text up to n tokens long.[14]

Next, the DCR model builds a representation for *each* of these candidate spans (or *chunks*). Suppose that we have a sequence (over context locations) of representations $\{\overrightarrow{\boldsymbol{b}_1}, \overleftarrow{\boldsymbol{b}_1}, \ldots, \overrightarrow{\boldsymbol{b}_N}, \overleftarrow{\boldsymbol{b}_N}\}$,

[12] http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

[13] When reading the paper, focus on the *Boundary Model* (described here), not the *Sequence Model*.

[14] If you implement this model you should consult your histogram of answer lengths in order to select a sensible value for n.

which we obtained as the forwards and backwards hidden states from a bidirectional RNN encoder (note we have not explained what the input to this bidirectional RNN was). To represent the candidate chunk from position i to position k, the DCR takes the concatenation $[\overrightarrow{\boldsymbol{b}_i}; \overleftarrow{\boldsymbol{b}_k}]$.

Once you've obtained these fixed-size representations for each candidate chunk, you can implement a simple softmax-based Output Layer similar to the one in our baseline – but instead it will produce a probability distribution over candidate chunks (instead of two distributions over context locations).

If you choose to implement this model, you will need to think about how you will build the chunk representations from the $\{\overrightarrow{\boldsymbol{b}_i}, \overleftarrow{\boldsymbol{b}_i}\}_{i=1}^{N}$ in TensorFlow. As one option, you could write code that, during the graph-building phase, loops over all candidate spans and defines the chunk representation by manually taking the correponding slices of the $\boldsymbol{b}$ tensor.

If you implement this model, think about the advantages and disadvantages of this approach, and comment on them in your writeup. How scalable is this model to longer pieces of text? What is the complexity of enumerating and processing all possible chunks? Can you make this model more efficient?

5.5 Additional input features

Appears in: DrQA [8].

Although Deep Learning has the ability to learn end-to-end, using the right input features can still boost performance significantly. For example, the DrQA model obtains high performance on SQuAD with an architecture as simple as our baseline, by including some useful input features.

In particular, the DrQA model uses several additional features (alongside word vectors) to represent a context token t_i, for $i \in \{1, \ldots, N\}$:

- *Exact Match.* Whether t_i also appears in the question.
- *Token Features.* The Part-of-Speech tag, the Named Entity type and the Normalized Term Frequency of t_i.
- *Aligned Question Embedding.* Use the word embedding for t_i to attend to the word embeddings for the question. Use the resulting attention output vector as an additional feature for t_i.

If you implement a model like this, reflect on the tradeoff between feature engineering and end-to-end learning. Feature engineering provides a significant benefit here; under what circumstances do you think feature engineering would be less successful?

5.6 Iterative reasoning (advanced)

Appears in: Dynamic Coattention Network [3], *Stochastic Answer Network* [9], *Gated Attention Reader* [10].

The core idea of iterative reasoning is that the network goes through several iterations of reasoning, potentially considering several answer options before settling on the right one. Each hop of reasoning depends on what was computed on the previous hop. These systems are generally more complex and may be more difficult to implement.

5.7 Other tips for improvement

There are many other things besides architecture changes that you can do to improve your performance. The suggestions in this section are mostly quick to implement, but it will take time to run the necessary experiments and draw the necessary comparisons. Remember that we will be grading your experimental thoroughness, so do not neglect the hyperparameter search!

- **Smarter span selection at test time**. At test time, our baseline takes separate argmaxes over p^{start} and p^{end} to get the predicted span – even if that means that the end location occurs before the start location. You could edit `QAModel.get_start_end_pos()` in `qa_model.py` to implement a better rule. For example, in the DrQA paper [8], they choose the start and end location pair (i, j) with $i <= j <= i + 15$ that maximizes $p^{\text{start}}(i)p^{\text{end}}(j)$.

- **Regularization**. The baseline code uses dropout. Experiment with different values of dropout and different types of regularization.
- **Sharing weights**. The baseline code uses the same RNN encoder weights for both the context and the question. This can be useful to enrich both the context and the question representations. Are there other parts of the model that could share weights? Are there conditions under which it's better to not share weights?
- **Word vectors**. By default, the baseline model uses 100-dimensional pre-trained GloVe embeddings to represent words, and these embeddings are held constant during training. You can experiment with other sizes or types of word embeddings, or try retraining the embeddings.
- **Combining forwards and backwards states**. In the baseline, we concatenate the forward and backward hidden states from the bidirectional RNN. You could try adding, averaging or max pooling them instead.
- **Types of RNN**. Our baseline uses a bidirectional GRU. You could try a LSTM instead.
- **Model size and number of layers**. With any model, you can try increasing the model size, usually at the cost of slower runtime.
- **Optimization algorithms**: The baseline uses the Adam optimizer. TensorFlow supports many other optimization algorithms. You might also experiment with learning rate annealing. You should also try varying the learning rate.
- **Ensembling**: Ensembling almost always boosts performance, so try combining several of your models together for your final submission. However, ensembles are more computationally expensive to run.

6 Alternative Goals

For most students, the primary goal of this assignment is to build a system that performs well at the main SQuAD task. However, listed here are some alternative goals you could pursue, instead of (or in addition to) the standard task. We would love to see some students attack these very interesting and important research problems:

- **Adversarial-proof SQuAD**: It was recently shown that state-of-the-art SQuAD models can be fooled by the placement of distracting sentences in the context paragraph [11]. This exposes a worrying weakness in the systems and techniques we use to solve SQuAD. Can you build a SQuAD model that is more robust against adversarial examples? To evaluate your model, you will consider not only F1 and EM score on the standard SQuAD test set, but also F1 and EM scores on the publicly-released test set of adversarial examples.[15]
- **Fast SQuAD**: Due to their recurrent nature, RNNs are slow, and they do not scale well when processing longer pieces of text. Can you find faster, perhaps non-recurrent deep learning solutions for SQuAD? As an example, there is a recent ICLR 2018 paper applying the Attention Is All You Need model (that we saw in lectures) to SQuAD.[16] If you want to focus on building a fast SQuAD model, you should review the non-recurrent and quasi-recurrent model families we saw in lectures. To evaluate your model, you will consider not only F1 and EM score, but also how fast (big-O computational complexity, and seconds per example in practice) you can generate answers.
- **Semi-supervised SQuAD**: The need for labeled data is arguably the greatest challenge facing Deep Learning today. Finding ways to get more from less data is an important research direction. How well can you perform on the SQuAD test set using only 30% of the training data? 20%? 10%? How might you use large amounts of unlabeled data to approach this task in an semi-supervised way? To evaluate your model, you will consider not only F1 and EM score after training on the full training set, but also F1 and EM score after training on only a fraction of the training set.

[15] https://worksheets.codalab.org/worksheets/0xc86d3ebe69a3427d91f9aaa63f7d1e7d/
[16] https://openreview.net/forum?id=B14TlG-RW

- **Low-storage SQuAD**: Deep Learning models typically have millions of parameters, which requires a large amount of storage space – this can be a problem, especially for mobile applications. Model compression is an active area of Deep Learning research [12], with many existing techniques for compression. Can you design a SQuAD model that requires less storage space? To evaluate your model, you will consider not only your F1 and EM score, but also the storage size of your model (in number of parameters, and in megabytes).

If you would like to work on these problems, you will need to submit your model for evaluation on CodaLab the same way as other teams. However, make it clear in your writeup that you are pursuing these alternative goals, and include the relevant performance metrics. We will grade your project holistically, taking into account your creativity, success and effort in pursuing the alternative goals.

7 Submitting to CodaLab

You will submit your models to CodaLab (`http://codalab.org`), where they will be evaluated against the dev set and the test set, and the results displayed on leaderboards. The dev set leaderboard (which you may submit to many times) will allow you to track your progress in comparison to your classmates. The test set leaderboard (which you may submit to only 3 times total) will provide the final F1 and EM numbers which will influence your grade. For full details, see the CodaLab submission instructions on the project page.

8 Grading Criteria

The final project will be graded holistically. This means we will look at many factors when determining your grade: the creativity, complexity and technical correctness of your approach, your thoroughness in exploring and comparing various approaches, the strength of your results, and the quality of your write-up, evaluation, and error analysis. Generally, more complicated improvements are worth more, and improvements we describe less in this handout are worth more because they require more creativity and research to implement. You are not required to pursue original ideas, but the best projects in this class will go beyond the improvements described in this document, and may in fact become published work themselves!

There is no expected F1 or EM score, nor expected number of improvements. Doing a small number of improvements with good results and thorough experimentation/analysis is better than implementing a large number of improvements that don't work, or barely work. In addition, the quality of your writeup and experimentation is important: we expect you to convincingly show your improvements are effective and describe why they work (and when they don't work).

In the analysis section of your report, we want to see you go beyond the simple F1 and EM results of your model. Try breaking down the scores – for example, how does your model perform on questions that start with 'who'? Questions that start 'when'? Questions that start 'why'? What are the other categories? Can you categorize the types of errors made by your model?

Larger teams are expected to do correspondingly larger projects. We will expect more improvements implemented, more thorough experimentation, and better results from teams with more members.

Note that only your F1/EM scores submitted to the **test** leaderboard will influence your grade – your submissions to the dev and sanity check leaderboards will not be taken into account.

9 Honor Code

We take Stanford's student honor code seriously. Here are honor code guidelines for the final project:

1. You are allowed to use whatever existing code, libraries, or data you wish, with one exception (see Rule 2). However, you must clearly cite your sources and indicate which parts of the project are not your work. If you use or borrow code from any external libraries, describe how you use the external code, and provide a link to the source in your writeup.

2. As an exception to Rule 1, you **may not** use a pre-existing implementation for the SQuAD challenge (for example, the publicly-available code for BiDAF) as your starting point unless you wrote that implementation yourself. If you believe you have a good reason to use a pre-existing SQuAD implementation as your starting point (for example, you have a specific cutting-edge research idea that would build on the state-of-the-art), then discuss with Richard/Abi/Kevin to get permission.

3. You are free to discuss ideas and implementation details with other teams (in fact, we encourage it!). However, under no circumstances may you look at another CS224n group's code, or incorporate their code into your project.

4. Do not share your code publicly (e.g., in a public github repo) until after the class has finished.

5. If you are doing a similar project for another class, you must make this clear and write down the exact portion of the project that is being counted for CS224n.

10 FAQs

10.1 How are out-of-vocabulary words handled?

Our baseline represents input words using pre-trained fixed GloVe embeddings. Out-of-vocabulary words (i.e. those that don't have a GloVe embedding) are represented by the UNK token. This is a special token in the vocabulary that has its own fixed word vector (set to a small random value).[17]

One way to improve the treatment of out-of-vocabulary words is to implement a character-level model (see Section 5.2).

10.2 How are padding and truncation handled?

In the baseline code, the hyperparameters `context_len` and `question_len` define the maximum length (in tokens) of the context and question respectively. These are 600 and 30 by default. The TensorFlow graph is initialized with these lengths pre-defined: meaning that, for example, the batches of context inputs are all padded up to length 600. We have a special PAD token in the vocabulary, with its own fixed word vector (which is set to a small random value) for this purpose.[18]

However, we also keep track of where the padding is (this is what `context_mask` is for) and ensure that we do not use the representations that correspond to padding. For example, when the context hidden states attend to the question hidden states, we mask before taking the softmax to obtain the attention distribution. This ensures that there is no attention on the padding parts of the question. Similarly, when we compute the softmax layer to get the start and end distributions, we mask to ensure that there is no probability mass on the padding parts of the context. In this way, the computations of the neural net are mathematically identical to what is described in Section 4.1, even though there is padding.

Inputs that are longer than `context_len` or `question_len` are discarded (when training) or truncated (at test time). You could try reducing `context_len` if you like – this will give you faster training iterations but you will throw away more training data, and at test time, you will truncate potentially important data.

If you wanted, you could change the code such that the length of the context and question are not pre-defined in the TensorFlow graph. This makes some things more difficult (for example, implementing the Dynamic Chunk Reader in Section 5.4), but it may make other things easier (you can unroll your RNNs for as many steps as necessary during training and testing).

10.3 Which parts of the code can I change?

When you submit your model to CodaLab, we will instruct you to run `main.py` with `--mode=official_eval`. This command loads your model from a checkpoint, loads the data from a JSON file, generates answers, and writes those answers to another JSON file. For more details, see the CodaLab submission instructions on the project page.

This is the only requirement of your code – you need to be able to run a command that reads the data from a JSON file and writes the answers to another JSON file. Therefore you could, if you wish, delete all the provided code and write your own from scratch – as long as you had some way to run the necessary command that does the required operation.

More likely, you will mostly change `qa_model.py` and `modules.py`. In this case, you just need to ensure that `QAModel.get_start_end_pos()` works correctly, because the function is called in `official_eval` mode. If you also change the way data is loaded (for example, `data_batcher.py`), then you may need to change some code in `official_eval_helper.py`, which loads the data from the JSON file in `official_eval` mode (though, if your preprocessing is light enough that you can place it in `qa_model.py`, then that may be easier and avoid duplicating code).

[17] You could change the code so that the UNK embedding is learned, if you like.
[18] You could change the code so that the PAD embedding is learned, if you like.

References

[1] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. *CoRR*, abs/1606.05250, 2016.

[2] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[3] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. *arXiv preprint arXiv:1611.01604*, 2016.

[4] Zhilin Yang, Bhuwan Dhingra, Ye Yuan, Junjie Hu, William W Cohen, and Ruslan Salakhutdinov. Words or characters? fine-grained gating for reading comprehension. *arXiv preprint arXiv:1611.01724*, 2016.

[5] Zhiguo Wang, Wael Hamza, and Radu Florian. Bilateral multi-perspective matching for natural language sentences. *arXiv preprint arXiv:1702.03814*, 2017.

[6] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. *arXiv preprint arXiv:1608.07905*, 2016.

[7] Yang Yu, Wei Zhang, Kazi Hasan, Mo Yu, Bing Xiang, and Bowen Zhou. End-to-end answer chunk extraction and ranking for reading comprehension. *arXiv preprint arXiv:1610.09996*, 2016.

[8] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051*, 2017.

[9] Xiaodong Liu, Yelong Shen, Kevin Duh, and Jianfeng Gao. Stochastic answer networks for machine reading comprehension. *arXiv preprint arXiv:1712.03556*, 2017.

[10] Bhuwan Dhingra, Hanxiao Liu, Zhilin Yang, William W Cohen, and Ruslan Salakhutdinov. Gated-attention readers for text comprehension. *arXiv preprint arXiv:1606.01549*, 2016.

[11] Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. *arXiv preprint arXiv:1707.07328*, 2017.

[12] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.