CS 224N Default Final Project: Question Answering

Due date: Friday, March 16th 2018 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. It is fine to take ideas from other papers on question answering, but you should acknowledge them in your write-up.

Please review any additional instructions posted on the assignment page at http://cs224n.stanford.edu/assignments.html. Submission instructions will be included there.

1 Overview

In this project, you will become a Deep Learning NLP researcher! You will develop a deep learning solution 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 questions so you can better understand anything from a class textbook to 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, a team for the first time submitted a model that 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 explicitly stated in the paragraph. This means SQuAD systems don't have to generate the answer text. Instead, 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⟩ tuple. You can explore the dataset and look at the outputs of systems submitted to the SQuAD leaderboard 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

Each SQuAD question has multiple answers from different crowd workers. The answers don't always agree, which is why "human performance" on the SQuAD leaderboard is not 100%. Performance is measured via two metrics: F1 and Exact Match (EM) score. Exact match measures the percentage of system outputs that match one of the ground truth (human) answers exactly.

This is a pretty harsh metric because if all the crowd workers answer a question with "Albert Einstein" and your system answers the question with "Einstein", the system will get no credit for the answer. F1 measures overlap between the system's answer and the closest human answer. In the "Einstein" example, the model would have 100% precision (its answer appears in the human answer) and 50% recall (it gets on of the two tokens in the human answer right) so the F1 score would be $2 \times \text{prediction} \times \text{recall}/(\text{precision} + \text{recall}) = 2 * 50 * 100/(100 + 50) = 66.67$. Given that the SQuAD dataset has three human-provided answers for each question, the maximum F1 score is taken across the human-provided answers. The F1 score over all questions in the dataset are averaged to get the final metric.

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. We also will provide a baseline that gets decent results on the dataset. Your job is to improve this baseline.

Later in the handout, we describe several improvements that are commonly used on SQuAD. They should all improve over the baseline if implemented correctly. Most of the improvements come from recent research papers. Some of the methods are described quite thoroughly in this handout. However, we don't go over everything, so you should refer to the original recent papers to figure out all the details. In rare cases a detail won't even be in the paper, and you will have to use your own judgement to decide what to do. We describe other improvements in less detail, although we do point to relevant papers. These will be worth a bit more in terms of grading because they require more investigation into the method (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 a completely new idea!

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 score or minimum number of extensions 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

First, you need to get on a machine with GPUs. For this, we encourage you to use Azure – please see the assignment page for a link to instructions how to set up your virtual machine[1]. Once you are on an appropriate machine, clone the project Github repository[2]:

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

Note: if you use Github to manage your fork of the project, keep your repository **private**.

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.
- `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 attention module, and a 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 into an embedding matrix.

2.2 Setup

Once you are on an appropriate machine and have cloned the project repository, run `get_started.sh`:

```
cd cs224n-win18-squad
./get_started.sh
```

Say yes when the script asks you to confirm that new packages will be installed. This script creates a new conda[3] environment for you called `squad`. You need to remember to activate your `squad` environment everytime you use the code.[4] To activate the `squad` environment, run

[1] The instructions will be released later in the quarter.

[2] We encourage you to use git so that you can easily integrate any changes that we might make to the code. If you are not familiar with git, this is a good resource to learn the basics: https://www.codecademy.com/learn/learn-git

[3] 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

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

```
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 we will use.
  - `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, each corresponding to a different example in the train set.
  - `glove.6B.zip`: Zipped GloVe vectors.
  - `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, please ask on Piazza for assistance.

3 The SQuAD Data

The SQuAD dataset has three splits: **train**, **dev** and **test**. The train and dev sets are publicly available to researchers, while the test set is kept secret. The test set is used to evaluate models submitted to the official leaderboard[5].

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[6], 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[7] – 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).

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

[5] https://rajpurkar.github.io/SQuAD-explorer/
[6] http://codalab.org
[7] Technically, the SQuAD dataset contains *three* human-provided reference answers for each question. However, for the purposes of training our SQuAD models, we just use the first human-provided answer.

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, **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).

RNN Encoder Layer: For each SQuAD example $(\text{context}_t, \text{question}_t, \text{answer}_t)$, the context is represented by a sequence of d-dimensional word embeddings $x_1, ..., x_N \in \mathbb{R}^d$, and the question by a sequence of d-dimensional word embeddings $y_1, ..., y_M \in \mathbb{R}^d$. These are fixed, pre-trained GloVe embeddings. The embeddings are fed into a 1-layer bidirectional GRU (which is shared between the context and the question):

$$\{\overrightarrow{c_1}, \overleftarrow{c_1}, ..., \overrightarrow{c_N}, \overleftarrow{c_N}\} = \text{biGRU}(\{x_1, ..., x_N\})$$
$$\{\overrightarrow{q_1}, \overleftarrow{q_1}, ..., \overrightarrow{q_M}, \overleftarrow{q_M}\} = \text{biGRU}(\{y_1, ..., y_M\})$$

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

$$c_i = [\overrightarrow{c_i}; \overleftarrow{c_i}] \in \mathbb{R}^{2h}$$
$$q_j = [\overrightarrow{q_j}; \overleftarrow{q_j}] \in \mathbb{R}^{2h}$$

Attention Layer: Next, we apply standard attention, with the context hidden states c_i attending to the question hidden states q_j. For each context hidden state c_i, the *attention distribution* $\alpha_i \in \mathbb{R}^M$ is computed as follows:

$$e_i = [c_i^T q_1, ..., c_i^T q_M] \in \mathbb{R}^M$$
$$\alpha_i = \text{softmax}(e_i) \in \mathbb{R}^M$$

The attention distribution is then used to take a weighted sum of the question hidden states q_j, producing the *attention output* a_i:

$$a_i = \sum_{j=1}^{M} \alpha_{ij} q_j \in \mathbb{R}^{2h}$$

The attention outputs are then concatenated to the context hidden states to obtain the *blended representations* b_i:

$$b_i = [c_i; a_i] \in \mathbb{R}^{4h}$$

Output Layer: Next, the blended representations b_i are fed through a fully connected layer followed by a ReLU non-linearity:

$$b'_i = \text{ReLU}(W_{FC} b_i + v_{FC}) \in \mathbb{R}^h$$

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

$$\text{logits}_i^{\text{start}} = w_{\text{start}}^T b'_i + u_{\text{start}} \in \mathbb{R}$$

where $w_{\text{start}} \in \mathbb{R}^h$ 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^{start} over the context locations $\{1, ..., N\}$:

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

We compute a probability distribution p^{end} in the same way (though with separate weights w_{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 $i_{\text{start}} \in \{1, ..., N\}$ and $i_{\text{end}} \in \{1, ..., N\}$ 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.

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 $(\ell^{\text{start}}, \ell^{\text{end}})$:

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

4.2 Train the baseline

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, you should see batches taking around 1.1 to 1.3 seconds on average. You should also see that there is a new directory in the `experiments` directory called `baseline`. This is where you can find all data relating to this experiment. In particular, you will 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

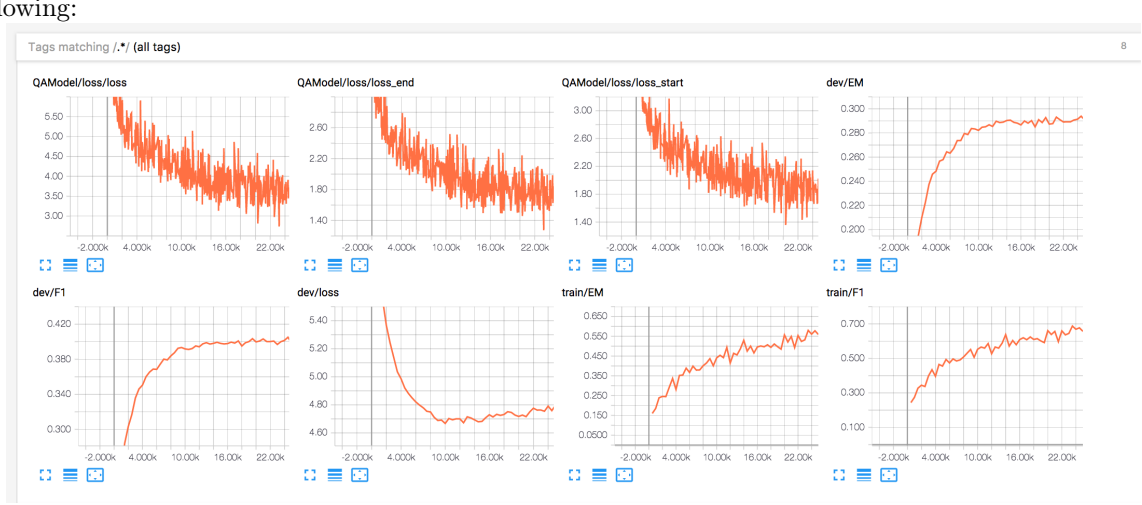
We strongly encourage you to use TensorBoard – it's one of the best features of TensorFlow and will enable you to get a much better view of your experiments. To use TensorBoard, navigate to the `experiments` directory and run:

```
tensorboard --logdir=. --port=5678
```

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. Now on your local machine, open http://localhost:1234/ in your browser. You should see something like the following:



In particular, you should see the training loss decreasing. (The F1 and EM plots may take some time to appear because they are logged less frequently). If you train the baseline for about 20k iterations or more, you should find that:

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

On a NV6 machine, 20k iterations should take around 6 or 7 hours.

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 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 try to 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 analyse 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 standard 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 standard Attention Layer.

Assume we have context hidden states $c_1, ..., c_N \in \mathbb{R}^{2h}$ and question hidden states $q_1, ..., q_M \in \mathbb{R}^{2h}$. We compute the *similarity matrix* $S \in \mathbb{R}^{N \times M}$, which contains a similarity score S_{ij} for each pair (c_i, q_j) of context and question hidden states.

$$S_{ij} = w_{\text{sim}}^T [c_i; q_j; c_i \circ q_j] \in \mathbb{R}$$

Here, $c_i \circ q_j$ is an elementwise product and $w_{\text{sim}} \in \mathbb{R}^{6h}$ 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 S to obtain attention distributions α_i, which we use to take weighted sums of the question hidden states q_j, yielding *C2Q attention outputs* a_i. In equations, this is:

$$\begin{aligned} \alpha_i &= \text{softmax}(S_{i,:}) \in \mathbb{R}^M \\ a_i &= \sum_{j=1}^{M} \alpha_{ij} q_j \in \mathbb{R}^{2h} \end{aligned}$$

Next, we perform Question-to-Context(Q2C) Attention. For each context location i, we take the max of the corresponding row of the similarity matrix $m_i = \max_j S_{ij}$. Then we take the softmax over the resulting vector $m \in \mathbb{R}^N$ – 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 question hidden states – this is the *Q2C attention output* c'. In equations:

$$\begin{aligned} m_i &= \max_j S_{ij} \in \mathbb{R} \forall i \\ \beta &= \text{softmax}(m) \in \mathbb{R}^N \\ c' &= \sum_{i=1}^{N} \beta_i c_i \in \mathbb{R}^{2h} \end{aligned} \tag{1}$$

Lastly, for each context location i we obtain the output b_i of the Attention Flow Layer by concatenating the context hidden state c_i, the C2Q attention output a_i, and the Q2C attention

output c':

$$b_i = [c_i, a_i, c'] \in \mathbb{R}^{6h}$$

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 standard Attention Layer.

Assume we have context hidden states $c_1, ..., c_N \in \mathbb{R}^l$ and question hidden states $q_1, ..., q_M \in \mathbb{R}^l$. First, apply a linear layer with tanh nonlinearity to the question hidden states to obtain projected question hidden states $q'_1, ..., q'_M$:

$$q'_j = \tanh(Wq_j + b) \in \mathbb{R}^l$$

Next, add *sentinel vectors* $c_\emptyset \in \mathbb{R}^l$ and $q'_\emptyset \in \mathbb{R}^l$ (which are a trainable parameters of the model) to both the context and the question states. This gives us $\{c_1, ..., c_N, c_\emptyset\}$ and $\{q'_1, ..., q'_M, q'_\emptyset\}$.

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

$$L_{ij} = c_i^T q'_j \in \mathbb{R}$$

Next, we use the affinity matrix L to compute standard attention in both directions.
For the Context-to-Question (C2Q) Attention, we obtain *C2Q attention outputs* a_i:

$$\alpha_i = \text{softmax}(L_{i,:}) \in \mathbb{R}^{M+1}$$
$$a_i = \sum_{j=1}^{M+1} \alpha_{ij} q'_j \in \mathbb{R}^l$$

For the Question-to-Context (Q2C) Attention, we obtain *Q2C attention outputs* b_i:

$$\beta_j = \text{softmax}(L_{:,j}) \in \mathbb{R}^{N+1}$$
$$b_j = \sum_{i=1}^{N+1} \beta_{ij} c_i \in \mathbb{R}^l$$

Next, we use the C2Q attention distributions α_i to take weighted sums of the Q2C attention outputs b_j. This gives us *second-level attention outputs* s_i:

$$s_i = \sum_{j=1}^{M+1} \alpha_{ij} b_j \in \mathbb{R}^l$$

Finally, we concatenate the second-level attention outputs s_i with the first-level C2Q attention outputs a_i, and feed the sequence through a bidirectional LSTM. The resulting hidden states u_i of this biLSTM are known as the *coattention encoding*. This is the overall output of the Coattention Layer.

$$\{u_1, ..., u_N\} = \text{biLSTM}(\{[s_1; a_1], ..., [s_N; a_N]\})$$

5.1.3 Self-attention

Appears in: r-net.[8]

We will not describe self-attention in detail here, as it was described in lectures. Self-attention (for example, the context representations attending to themselves) could be a useful technique to use *in addition to* attention between the context and question. This is what the r-net model does.

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

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 popular. 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.

We usually start by representing the characters $c_1, ..., c_L$ using trainable *character embeddings* $e_1, ..., e_L$. You can think of character embeddings are analogous to word embeddings, though character embeddings typically have smaller dimension, 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 $e_1, ..., e_L \in \mathbb{R}^d$, and compute another sequence of hidden representations $h_1, ..., h_L \in \mathbb{R}^f$. The core idea is that each h_i is computed based on a *window* of characters $[c_{i-w}; ...; c_i; ...; c_{i+w}]$ centered at position i (w is the window width). Finally, you apply *max pooling* to obtain the character-level encoding: $\text{emb}_{\text{char}}(w) = \max_i h_i$.

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` 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 h_i) here, and `kernel_size`, which corresponds to w (window width) here. You could also construct a character-CNN with several layers (perhaps with different f and w on each layer). You can learn more about CNNs for NLP here[9].

5.3 Conditioning End Prediction on Start Prediction

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

Our baseline predicts the start location and the end location *independently*, given the final layer's activations b'. Instead, you could build a model that explicitly conditions the calculation of the end location on the predicted start location.

The Answer Pointer component of the 'Match-LSTM with Answer Pointer' model[10] is one such example. The Answer Pointer is similar to the decoder component of a sequence-to-sequence model (that we learned about in lectures). That is: the Answer Pointer is a RNN that is run for exactly two steps. On the first step, it produces p^{start}. We take the argmax over p^{start} to get the predicted start location $\ell^{\text{start}} = \text{argmax}_i p_i^{\text{start}}$. Then ℓ^{start} is fed as input into the next step of the RNN, which produces p^{end}. Finally we take argmax over p^{end} to get ℓ^{end}.

Note: taking argmax over p^{start} and feeding the result into the second step of the RNN is non-differentiable, so we can't do this during training. So we do something slightly different during training: on the second step of the RNN we feed the *true* ℓ^{start} as input to the RNN. This is also how RNN language models, and sequence-to-sequence models are trained. The technique is called *teacher forcing*.

You could implement a system like this to replace the Output Layer of our baseline.

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

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

5.4 Span Representations

Appears in: Dynamic Chunk Reader [7].

The SQuAD task can be phrased as the task of determining the 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) implicitly break down this probability:

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

and compute the probabilities for the start and end locations separately.

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[11].

Next, the DCR model builds a representation for *each* of these candidate spans (or *chunks*). Assume that we have a sequence (over context locations) of blended hidden states $\{\overrightarrow{b'_1}, \overleftarrow{b'_1}, ..., \overrightarrow{b'_N}, \overleftarrow{b'_N}\}$, with forwards and backwards directions from a bidirectional RNN encoder. To represent the candidate chunk from position i to position k, the DCR takes the concatenation $[\overrightarrow{b'_i}; \overleftarrow{b'_k}]$.

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 $\{\overrightarrow{b'_i}, \overleftarrow{b'_i}\}_{i=1}^{N}$ 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 b' tensor.

If you implement this model, think about the advantages and disadvantages of this approach, and comment on them in your writeup. Is this model scalable 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 $i \in \{1, ..., N\}$:

- *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.* Compute standard attention from the context word embeddings to the question word embeddings. Use the resulting attention output vectors as additional features for the context tokens.

If you implement a model like this, reflect on the tradeoff between feature engineering and end-to-end learning. Under what circumstances do you think feature engineering would be less successful?

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

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. 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 shares the RNN encoder weights between 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 expense 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 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 goal of this assignment is to build a system that performs well at the main SQuAD task. However, listed here are some alternative tasks 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 [11] that state-of-the-art SQuAD models can be fooled by the placement of distracting sentences in the context paragraph. This exposes a worrying weakness in the systems and techniques we use to solve SQuAD. You

could aim to build a SQuAD model that is more robust against adversarial examples, and evaluate it against the publicly-released test set of adversarial examples[12].

- **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.[13] 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 would consider not only F1 and EM score, but also how fast (seconds per example) 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 high a score can you get 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?
- **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. How small a model can you produce (in number of parameters, or in MB) and still get a good score? Model compression is an active area of Deep Learning research [12].

Further details for these alternative goals (e.g. submission requirements and grading criteria) will be announced at a later date. For now, if you're interested in working on these problems, then just submit your project team in the same way as general Default Final Project teams.

7 Submitting to CodaLab

You will submit your models to CodaLab[14], 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. More details about the submission process will be released later in the quarter.

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 extensions will be worth more, and extensions we describe less in this handout will be worth a little bit extra because they require more creativity and research to implement.

There is no expected F1 or EM score, nor expected number of extensions. Doing a small number of extensions with good results and thorough experimentation/analysis is better than implementing a large number of extensions 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 extensions are effective and describe why the work (and when they don't work).

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

9 Honor Code

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

[12] https://worksheets.codalab.org/worksheets/0xc86d3ebe69a3427d91f9aaa63f7d1e7d/
[13] https://openreview.net/forum?id=B14TlG-RW
[14] http://codalab.org

- You are allowed to use whatever existing code, libraries, or data you wish. 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 what you use the external code for and provide a link to the source in your writeup.
- 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.
- Do not share your code publicly (e.g., in a public github repo) until after the class has finished.
- 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.

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.