Homework 2: Question Answering on SQuAD 2.0

Last updated on Oct. 27, 2019

Contents

1 Overview

In the homework 2 (HW2), you will explore deep learning techniques for question answering on the Stanford Question Answering Dataset (SQuAD) [1]. Homework is designed to enable you to dive right into deep learning experiments without spending too much time getting set up. You will have the chance to implement current state-of-the-art techniques and experiment with your own novel designs. This homework will use the updated version of SQuAD, named *SQuAD 2.0* [2], which extends the original dataset with unanswerable questions.

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 legal document.

SQuAD is less than three 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. At the top you will see models for SQuAD 2.0 (the version we will be using). Notice how the leaders surpassed human performance on SQuAD 2.0. Also notice that the leaderboard is extremely active, with first-place submissions appearing in mid-Sep. 2019.

The paragraphs in SQuAD are from Wikipedia. The questions and answers were crowdsourced using Amazon Mechanical Turk. There are around 150k questions in total, and *roughly half of the questions cannot be answered using the provided paragraph* (this is new for SQuAD 2.0). However, if the question *is* answerable, the answer is a chunk of text taken directly from the paragraph. This means that 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/v2.0/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 official dev and test set, every answerable 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: **Exact Match (EM)** score and **F1** 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 $2 \times \text{prediction} \times \text{recall}/(\text{precision}+\text{recall}) = 2*50*100/(100+50) = 66.67\%$.

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

- When a question has no answer, both the F1 and EM score are 1 if the model predicts no-answer, and 0 otherwise.
- For questions that do have answers, when evaluating on the dev or test sets, we take the *maximum* F1 and EM scores across the three human-provided answers for that question. 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.

1.2 This homework

The goal of this homework is to produce a question answering 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 models and techniques that are commonly used in high-performing SQuAD models – most come from recent research papers. We provide these suggestions to help you get started implementing better models. They should all improve over the baseline if implemented correctly (and note that there is usually more than one way to implement something correctly).

Though you're not required to implement something original, the best returns will (and in fact may become research papers in the future). Originality 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 quantitatively 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.

The homework is open-ended – it will be up to you to figure out what to do. 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.

For more information on grading criteria, see Section 7.

2 Getting Started

For this homework, you will need a machine with GPUs to train your models efficiently. For this, ask help from other classmates if you do not have a GPU.

We advise that you **develop your code on your local machine**, using PyTorch without GPUs, and move to your classmates's machine only once you've debugged your code and you're ready to train. Note: If you use GitHub to manage your code, you must keep your repository **private**.

When you work through this *Getting Started* section for the first time, do so on your local machine. You will then repeat the process on your classmates's machine .

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

```
git clone https://github.com/chrischute/squad.git
```

This repository contains the starter code and the version of SQuAD that we will be using. We encourage you to `git clone` our repository, rather than simply downloading 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 `squad` directory and run the `git pull` command.

2.1 Code overview

The repository `squad` contains the following files:

- `args.py`: Command-line arguments for `setup.py`, `train.py`, and `test.py`.
- `environment.yml`: List of packages in the `conda` virtual environment.
- `layers.py`: Layers used by the models.
- `models.py`: The starter model, and any others you might add.
- `setup.py`: Downloads pretrained GloVe vectors and preprocesses the data.
- `train.py`: Top-level entrypoint for training the model.
- `test.py`: Top-level entrypoint for testing the model and generating submissions for the leaderboard.
- `util.py`: Utility functions and classes.

In addition, you will notice two directories:

- `data/`: Contains our custom SQuAD dataset, both the unprocessed JSON files, and (after running `setup.py`), all preprocessed files.
- `save/`: Location for saving all checkpoints and logs. For example, if you train the baseline with `python train.py -n baseline`, then the logs, checkpoints, and TensorBoard events will be saved in `save/train/baseline-01`. The suffix number will increment if you train another model with the same name.

2.2 Setup

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

- Make sure you have Anaconda or Miniconda (https://conda.io/docs/user-guide/install/index.html#regular-installation) installed.
  - **Conda** is a package manager that sandboxes your projects dependencies in a virtual environment.
  - **Anaconda** contains Conda, plus many other data science packages.
  - **Miniconda** is more minimal than Anaconda; it contains Conda and its dependencies and no extra packages by default.
- `cd` into `squad` and run `conda env create -f environment.yml`
  - This creates a Conda environment called `squad`.
- Run `source activate squad`
  - This activates the `squad` environment.
  - **Remember to do this each time you want to work on or use your code!**
- Run `python setup.py`
  - This downloads GloVe 300-dimensional word vectors, and the SQuAD 2.0 training and dev sets.
  - This also pre-processes the dataset for efficient data loading.
  - For a MacBook Pro, `setup.py` takes around 30 minutes total.
- *(Optional)* If you would like to use PyCharm, select the `squad` environment. Example instructions for Mac OS X:
  - Open the `squad` directory in PyCharm.
  - Go to `PyCharm > Preferences > Project > Project interpreter`.
  - Click the gear in the top-right corner, then `Add`.
  - Select `Conda environment > Existing environment > Click '...' on the right`.
  - Select `/Users/YOUR_USERNAME/miniconda3/envs/squad/bin/python`.
  - Select `OK` then `Apply`.

Once the `setup.py` script has finished, you should now see many additional files in `squad/data`:

- `{train,dev,test}-v2.0.json`: The official SQuAD train set, and our modified version of the SQuAD dev and test sets. See Section 3 for details. Note that the test set does not come with answers.
- `{train,dev,test}_{eval,meta}.json`: Tokenized training and dev set data.
- `glove.840B.300d/glove.840B.300d.txt`: Pretrained GloVe vectors. These are 300-dimensional embeddings trained on the CommonCrawl 840B corpus.
- `{word,char}_emb.json`: Word and character embeddings, where we kept only the words and characters that appear in the training set. This trimming process is common practice to reduce the size of the embedding matrix and free up memory for your model.
- `{word,char}2idx.json`: Dictionaries mapping character and words (strings) to indices (integers) in the embedding matrices in `{word,char}_emb.json`.

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 `setup.py` for error messages, and ask for assistance on QQ if applicable.

3 The SQuAD Data

3.1 Data splits

The official SQuAD 2.0 dataset has three splits: **train**, **dev** and **test**. The train and dev sets are publicly available and the test set is entirely secret. To compete on the official SQuAD leaderboards, researchers submit their models, and the SQuAD team runs the models on the secret test set.

For simplicity and scalability, we are instead running our class leaderboard 'Kaggle-style', i.e., we release test set's (context, question) pairs to students, and they submit their model-produced answers in a CSV file. We then compare these CSV files to the true test set answers and report scores in a leaderboard. Clearly, we cannot release the official test set's (context, question) pairs because they are secret. Therefore in this homework, we will be using custom dev and test sets, which are obtained by splitting the official dev set in half.

Given that the official SQuAD dev set contains our test set, **you must make sure not to use the official SQuAD dev set in any way**. You may *only* use our training set and our dev set to train, tune and evaluate your models. If you use the official SQuAD dev set to train, tune or evaluate your models, or to modify your CSV solutions in any way, you are committing an honor code violation. To detect cheating of this kind, we have produced a small amount of new SQuAD 2.0 examples whose answers are not publicly available, and added them to our test set – your relative performance on these examples, compared to the rest of our test set, would reveal any cheating. If you always use the provided GitHub repository and `setup.py` script to set up your SQuAD dataset, and don't use the official SQuAD dev set at all, you will be safe.

To summarize, we have the following splits:

- `train` (129,941 examples): All taken from the official SQuAD 2.0 training set.
- `dev` (6078 examples): Roughly half of the official dev set, randomly selected.
- `test` (5915 examples): The remaining examples from the official dev set, plus hand-labeled examples.

From now on we will refer to these splits as 'the train set', 'the dev set' and 'the test set', and always refer to the official splits as 'the official train set', 'the official dev set', and 'the official test set'.

You will use the train set to train your model and the dev set to tune hyperparameters and measure progress locally. Finally, you will submit your test set solutions to a class leaderboard, which will calculate and display your scores on the test set – see Section 6 for more information.

3.2 Terminology

The SQuAD dataset contains many (context, question, answer) triples[2] – 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.

[2] 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.

4 Training the Baseline

As a starting point, we have provided you with the complete code for a 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

The baseline model is a based on BiDAF (which is short for Bidirectional Attention Flow [3]). Unlike the original BiDAF model, our implementation does not include a character-level embedding layer. It may be a useful preliminary exercise to extend the baseline model to match the 'BiDAF-No-Answer (single model)' baseline score in last place on the official SQuAD 2.0 leaderboard, although we you should aim higher for your final project goal. In `model.py`, you will see that BiDAF follows the high-level structure outlined in the sections below. Throughout let N be the length of the context, let M be the length of the question, let D be the embedding size, and let H be the hidden size of the model.

Embedding Layer (`layers.Embedding`)

Given some input word indices[3] $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_k \in \mathbb{N}$, the embedding layer performs an embedding lookup to convert the indices into word embeddings $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{R}^D$. This is done for both the context and the question, producing embeddings $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_N \in \mathbb{R}^D$ for the context and $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_M \in \mathbb{R}^D$ for the question.

In the embedding layer, we further refine the embeddings with the following two step process:

1. We project each embedding to have dimensionality H: Letting $\boldsymbol{W}_{\text{proj}} \in \mathbb{R}^{H \times D}$ be a learnable matrix of parameters, each embedding vector $\boldsymbol{v}_i$ is mapped to $\boldsymbol{h}_i = \boldsymbol{W}_{\text{proj}} \boldsymbol{v}_i \in \mathbb{R}^H$.

2. We apply a Highway Network [4] to refine the embedded representation. Given an input vector $\boldsymbol{h}_i$, a one-layer highway network computes

$$\begin{aligned} \boldsymbol{g} &= \sigma(\boldsymbol{W}_g \boldsymbol{h}_i + \boldsymbol{b}_g) \in \mathbb{R}^H \\ \boldsymbol{t} &= \text{ReLU}(\boldsymbol{W}_t \boldsymbol{h}_i + \boldsymbol{b}_t) \in \mathbb{R}^H \\ \boldsymbol{h}'_i &= \boldsymbol{g} \odot \boldsymbol{t} + (1 - \boldsymbol{g}) \odot \boldsymbol{h}_i \in \mathbb{R}^H, \end{aligned}$$

   where $\boldsymbol{W}_g, \boldsymbol{W}_t \in \mathbb{R}^{H \times H}$ and $\boldsymbol{b}_g, \boldsymbol{b}_t \in \mathbb{R}^H$ are learnable parameters ($\boldsymbol{g}$ is for 'gate' and $\boldsymbol{t}$ is for 'transform'). We use a two-layer highway network to transform each hidden vector $\boldsymbol{h}_i$, which means we apply the above transformation twice, each time using distinct learnable parameters.

Note: The original BiDAF model uses learned character-level word embeddings in addition to the word-level embeddings used here. See Section 5 for an explanation of how one might add character-level embeddings.

Encoder Layer (`layers.RNNEncoder`)

The encoder layer uses a bidirectional LSTM [5] to allow the model to incorporate temporal dependencies between timesteps of the embedding layer's output. The encoded output is the RNN's hidden state at each position:

$$\begin{aligned} \boldsymbol{h}'_{i,\text{fwd}} &= \text{LSTM}(\boldsymbol{h}'_{i-1}, \boldsymbol{h}_i) \in \mathbb{R}^H \\ \boldsymbol{h}'_{i,\text{rev}} &= \text{LSTM}(\boldsymbol{h}'_{i+1}, \boldsymbol{h}_i) \in \mathbb{R}^H \\ \boldsymbol{h}'_i &= [\boldsymbol{h}'_{i,\text{fwd}}; \boldsymbol{h}'_{i,\text{rev}}] \in \mathbb{R}^{2H}. \end{aligned}$$

Note in particular that $\boldsymbol{h}'_i$ is of dimension $2H$, as it is the concatenation of forward and backward hidden states at timestep i.

[3] A *word index* is an integer that tells you which row (or column) of the embedding matrix contains the word's embedding. The `word2idx` dictionary maps words to their indices.

Attention Layer (`layers.BiDAFAttention`)

The core part of the BiDAF model is the **BiD**irectional **A**ttention **F**low 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.

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. In the starter code, the `get_similarity_matrix` method of the `layers.BiDAFAttention` class is a memory-efficient implementation of this operation. We encourage you to walk through the implementation of `get_similarity_matrix` and convince yourself that it indeed computes the similarity matrix as described above.

Next, we perform Context-to-Question (C2Q) Attention. We take the row-wise softmax of $\boldsymbol{S}$ to obtain attention distributions $\bar{\boldsymbol{S}}$, 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:

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

Next, we perform Question-to-Context(Q2C) Attention. We take the softmax of the columns of $\boldsymbol{S}$ to get $\bar{\bar{\boldsymbol{S}}} \in \mathbb{R}^{N \times M}$, where each column is an attention distribution over context words. Then we multiply with $\bar{\boldsymbol{S}}$ into $\bar{\bar{\boldsymbol{S}}}$, and use the result to take weighted sums of the hidden states $\boldsymbol{c}_j$ to get the *Q2C attention output*:

$$\begin{aligned}
\bar{\bar{\boldsymbol{S}}}_{:,j} &= \text{softmax}(\bar{\boldsymbol{S}}_{:,j}) \in \mathbb{R}^N \quad \forall j \in \{1, \ldots, M\} \\
\boldsymbol{S}' &= \bar{\boldsymbol{S}} \bar{\bar{\boldsymbol{S}}}^T \in \mathbb{R}^{N \times N} \\
\boldsymbol{b}_i &= \sum_{j=1}^{N} \boldsymbol{S}'_{i,j} \boldsymbol{c}_j \in \mathbb{R}^{2H} \quad \forall i \in \{1, \ldots, N\}.
\end{aligned}$$

Lastly, for each context location $i \in \{1, \ldots, N\}$ we obtain the output $\boldsymbol{g}_i$ 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 $\boldsymbol{b}_i$:

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

where $\circ$ represents elementwise multiplication.

Modeling Layer (`layers.RNNEncoder`)

The modeling layer is tasked with refining the sequence of vectors after the attention layer. Since the modeling layer comes after the attention layer, the context representations are conditioned on the question by the time they reach the modeling layer. Thus the modeling layer integrates temporal information between context representations conditioned on the question. Similar to the Encoder layer, we use a bidirectional LSTM Given input vectors $\boldsymbol{g}_i \in \mathbb{R}^{8H}$, the modeling layer computes

$$\begin{aligned}
\boldsymbol{m}_{i,\text{fwd}} &= \text{LSTM}(\boldsymbol{m}_{i-1}, \boldsymbol{g}_i) \in \mathbb{R}^H \\
\boldsymbol{m}_{i,\text{rev}} &= \text{LSTM}(\boldsymbol{m}_{i+1}, \boldsymbol{g}_i) \in \mathbb{R}^H \\
\boldsymbol{m}_i &= [\boldsymbol{m}_{i,\text{fwd}}; \boldsymbol{m}_{i,\text{rev}}] \in \mathbb{R}^{2H}.
\end{aligned}$$

The Modeling layer differs from the Encoder layer in that we use a one-layer LSTM in the Encoder layer, whereas we use a two-layer LSTM in the Modeling layer.

Output Layer (`layers.BiDAFOutput`)

The output layer is tasked with producing a vector of probabilities corresponding to each position in the context: $\boldsymbol{p}_{\text{start}}, \boldsymbol{p}_{\text{end}} \in \mathbb{R}^N$. As the notation suggests, $\boldsymbol{p}_{\text{start}}(i)$ is the predicted probability that the answer span starts at position i, and similarly $\boldsymbol{p}_{\text{end}}(i)$ is the predicted probability that the answer span ends at position i. (See the 'Predicting no-answer' section below for details on no-answer predictions).

Concretely, the output layer takes as input the attention layer outputs $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_N \in \mathbb{R}^{8H}$ and the modeling layer outputs $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N \in \mathbb{R}^{2H}$. The output layer applies a bidirectional LSTM to the modeling layer outputs, producing a vector $\boldsymbol{m}'_i$ for each $\boldsymbol{m}_i$ given by

$$\begin{aligned}
\boldsymbol{m}'_{i,\text{fwd}} &= \text{LSTM}(\boldsymbol{m}'_{i-1}, \boldsymbol{m}_i) \in \mathbb{R}^H \\
\boldsymbol{m}'_{i,\text{rev}} &= \text{LSTM}(\boldsymbol{m}'_{i+1}, \boldsymbol{m}_i) \in \mathbb{R}^H \\
\boldsymbol{m}'_i &= [\boldsymbol{m}'_{i,\text{fwd}}; \boldsymbol{m}'_{i,\text{rev}}] \in \mathbb{R}^{2H}.
\end{aligned}$$

Now let $\boldsymbol{G} \in \mathbb{R}^{8H \times N}$ be the matrix with columns $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_N$, and let $\boldsymbol{M}, \boldsymbol{M'} \in \mathbb{R}^{2H \times N}$ similarly be matrices with columns $\boldsymbol{m}_1 \ldots, \boldsymbol{m}_N$ and $\boldsymbol{m}'_1, \ldots, \boldsymbol{m}'_N$, respectively. To finally produce $\boldsymbol{p}_{\text{start}}$ and $\boldsymbol{p}_{\text{end}}$, the output layer computes

$$\boldsymbol{p}_{\text{start}} = \text{softmax}(\boldsymbol{W}_{\text{start}}[\boldsymbol{G}; \boldsymbol{M}]) \qquad \boldsymbol{p}_{\text{end}} = \text{softmax}(\boldsymbol{W}_{\text{end}}[\boldsymbol{G}; \boldsymbol{M}']),$$

where $\boldsymbol{W}_{\text{start}}, \boldsymbol{W}_{\text{end}} \in \mathbb{R}^{1 \times 10H}$ are learnable parameters. In the code, notice that the softmax operation uses the context mask, and we compute all probabilities in log-space for numerical stability and because the `F.nll_loss` function expects log-probabilities.

Training Details

Loss Function

Our loss function is the sum of the negative log-likelihood (cross-entropy) loss for the start and end locations. That is, if the gold start and end locations are $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, N\}$ respectively, then the loss for a single example is:

$$\text{loss} = -\log \boldsymbol{p}_{\text{start}}(i) - \log \boldsymbol{p}_{\text{end}}(j)$$

During training, we average across the batch and use the Adadelta optimizer [6] to minimize the loss.

Discretized Predictions

At test time, we discretize the soft predictions of the model to get start and end indices. We choose the pair (i, j) of indices that maximizes $\boldsymbol{p}_{\text{start}}(i) \cdot \boldsymbol{p}_{\text{end}}(j)$ subject to $i \leq j$ and $j - i + 1 \leq L_{\text{max}}$, where L_{max} is a hyperparameter which sets the maximum length of a predicted answer. We set L_{max} to 15 by default. Code can be found in the `discretize` function in `util.py`.

Predicting no-answer

To allow our model to make no-answer predictions, we adopt an approach that was originally introduced in Section 5 of [7]. In particular, we prepend a OOV token to the beginning of each sequence. The model outputs p_{start} and p_{end} soft-predictions as usual, so no adaptation is needed within the model. When discretizing a prediction, if $p_{\text{start}}(0) \cdot p_{\text{end}}(0)$ is greater than any predicted answer span, the model predicts no-answer. Otherwise the model predicts the highest probability span as usual. We keep the same NLL loss function.

Intuitively, this approach allows the model to predict a per-example confidence score that the question is unanswerable. If the model is highly confident that there is no answer, we predict no answer. In all cases the model continues to predict the most likely span if that answer exists.

Exponential Moving Average of Parameters

As recommended in the BiDAF paper, we use an exponentially weighted moving average of the model parameters during evaluation (with decay rate 0.999). Intuitively, this is similar to using an ensemble of multiple checkpoints sampled from one training run. The details can be found in the `util.EMA` class, and you will notice calls to `ema.assign` and `ema.resume` in `train.py`. It is worth experimenting with removing the exponential moving average or changing the decay rate when you train your own models.

4.2 Train the baseline

Before starting to train the baseline on your machine, consider opening a new session with `tmux` or some other session manager (https://github.com/tmux/tmux/wiki). This will make it easier for you to leave your model training for a long time, then retrieve the session later.

To start training the baseline, run the following commands:

```
source activate squad        # Activate the squad environment
python train.py -n baseline  # Start training
```

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

```
20\%|###             | 26112/129941 [02:53<09:48, 176.40it/s, NLL=6.54, epoch=1]
```

You should see the loss – shown as `NLL` for negative log-likelihood – begin to drop. On a single GPU instance, you should expect training to take about 22 minutes per epoch. Note that the starter code will automatically use more than one GPU if your machine has more available.

You should also see that there is a new directory under `save/train/baseline-01`. This is where you can find all data relating to this experiment. In particular, you will (eventually) see:

- `log.txt`: A record of all information logged during training. This includes a complete printout of the arguments at the very top, which can be useful when trying to reproduce results.
- `events.out.tfevents.*`: These files contain information (like the loss over time), which our code has logged so it can be visualized by TensorBoard.
- `step_N.pth.tar`: These are checkpoint files, that contain the weights of the model at checkpoints which achieved the highest validation metrics. The number `N` corresponds to how many training iterations had been completed when the model was saved. By default a checkpoint is saved every 50,000 iterations, but you can save checkpoints more frequently by changing the `eval_steps` flag.
- `best.pth.tar`: The best checkpoint throughout training. The metric used to determine which checkpoint is 'best' is defined by the `metric_name` flag. Typically you will load this checkpoint for use by `test.py`, which you can do by setting the `load_path` flag.

4.3 Tracking progress in TensorBoard

We strongly encourage you to use TensorBoard, as it will enable you to get a much better view of your experiments. To use TensorBoard, run the following command from the `squad` directory:

```
tensorboard --logdir save --port 5678   # Start TensorBoard
```

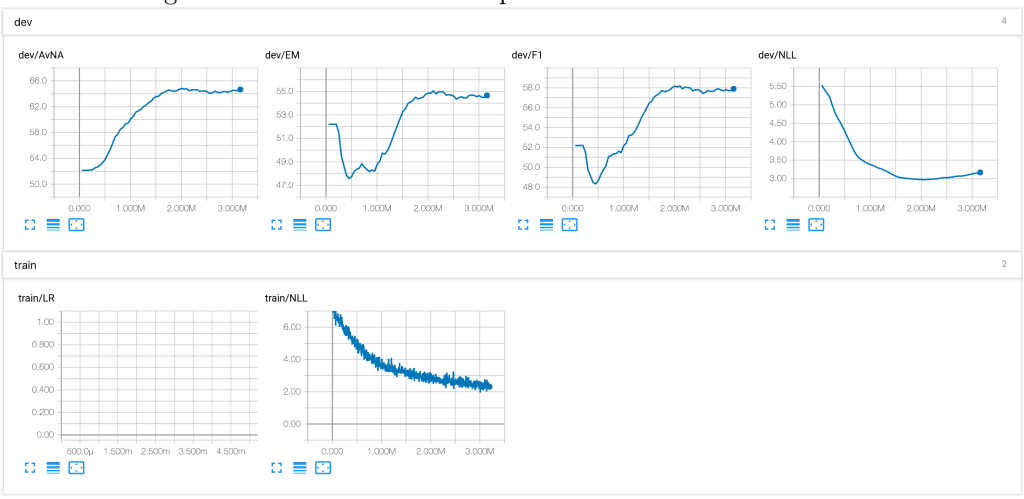
If you are training on your local machine, now open http://localhost:5678/ in your browser. If you are training on a remote machine , then run the following command on your local machine:

```
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, AvNA, EM, and F1 for both train and dev sets. EM and F1 are the official SQuAD evaluation metrics, and AvNA is a useful metric we added for debugging purposes. In particular, AvNA stands for **A**nswer **v**s. **N**o **A**nswer and it measures the classification accuracy of your model when only considering its answer (any span predicted) vs. no-answer predictions.

The dev plots may take some time to appear because they are logged less frequently than the train plots. However, you should see training set loss decreasing from the very start. Here is the view after training the baseline model for 25 epochs:



In particular, over 3 million iterations we find that:

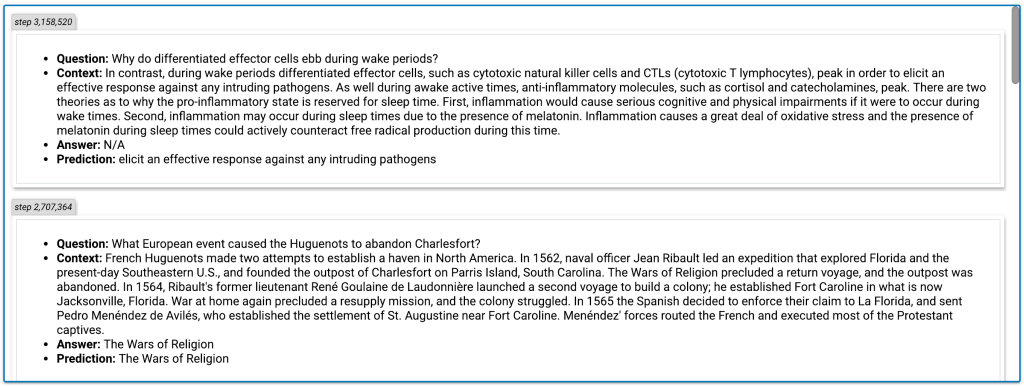
- The train loss continues to improve throughout
- The dev loss begins to rise around 2M iterations (overfitting)
- The dev AvNA reaches about 65, the dev F1 reaches about 58, and the dev EM score reaches around 55.
- Although the dev NLL improves throughout the training period, the dev EM and F1 scores initially get worse at the start of training, before then improving. We elaborate on this point below.

Regarding the last bullet point, this does not necessarily indicate a bug, but rather can be explained because we directly optimize the NLL loss, not F1 or EM: Early in training, the NLL is quickly reduced by always predicting no-answer. Since roughly half of the SQuAD examples are no-answer, a model predicting all no-answer will get close to 50% AvNA. In addition, the SQuAD 2.0 metrics define F1 and EM for no-answer examples to be 1 if the model predicts no answer and 0 otherwise. If we assume the model gets 0 F1 and EM on answerable examples, this results in a mean F1/EM score of roughly 50% very early in training.

We advise you to reproduce this experiment, i.e., train the baseline and obtain results similar to those we report above. 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 train over time, compared to the baseline.

4.4 Inspecting Output

During training you will also notice a tab in TensorBoard labeled `Text`. Try clicking on this tab and you should see output similar to the following:



Viewing these examples can be extremely helpful to debug your model, understand its strengths and weaknesses, and as a starting point for your analysis in your final report.

5 More SQuAD Models and Techniques

From here, the project is open-ended! As explained in Section 1.2, in this section we provide you with an overview of models, model families, and other common techniques that are used in high-performing SQuAD systems. Your job is to read about some of these models and techniques, understand them, choose some to implement, carefully train them, and analyze their performance – ultimately building the best SQuAD system you can. Implementation is an open-ended task: there are multiple valid implementations of a single model, and sometimes a paper won't even provide all the details – requiring you to make some decisions by yourself. To learn more about homework expectations and grading, see Section 8.

5.1 Pre-trained Contextual Embeddings (PCE), aka ELMo & BERT

If you glance at the SQuAD leaderboard, you may notice that almost every leading submission has a Sesame Street character in its name (ELMo [8] or BERT [9]). ELMo stands for **E**mbeddings from **L**anguage **Mo**dels and BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. The core idea of ELMo and BERT is that to represent a piece of text, we should use word embeddings that depend on the context in which the word appears in the text. In practice, this is achieved by pretraining weights on a large-scale language modeling dataset, then loading the pretrained weights to use as the first several layers of your model.

This year our leaderboard will have two distinct divisions: one for models that use pretrained contextual embeddings (PCE), and another for non-PCE models only (note that word vectors like word2vec, GloVe and FastText are allowed in the non-PCE division). We decided on this distinction primarily for two reasons:

1. Teams that use PCE will need to download code and import pretrained weights, meaning that large portions of their code will be externally-sourced. For these teams, we will focus our evaluation on what the team added on top of ELMo/BERT. By contrast, non-PCE teams are more likely to have mostly self-written code.
2. PCE approaches are likely to outperform the best non-PCE models by a large margin. This is reflected on the official SQuAD 2.0 leaderboard, where at least the top 15 entries all use BERT in some way. Though high-performing, the act of loading ELMo/BERT alone is an uncreative endeavor within the context of this homework. We want to ensure that teams who choose *not* to use PCE can still be competitive on the leaderboard. We hope that hosting a non-PCE division will encourage many teams to focus on more creative ways to boost performance.

If you do want to use PCE, keep in mind that simply loading e.g. BERT and getting a high score will not necessarily earn you a high grade on the homework. It's up to you to think creatively about how to improve upon a standard BERT implementation.

5.1.1 ELMo

Original paper: Deep contextual word representations [8]

In traditional word embeddings such as Word2Vec [10], GloVe [11], and FastText [12], each word in the vocabulary is mapped to a fixed vector, regardless of its context. The core idea of ELMo is to address this weakness by using the context in which a word appears to construct the embedding. In practice, ELMo implements this idea by training a two-layer bidirectional LSTM for language modeling on a large-scale corpus. This pretrained bi-LSTM can then be used as the embedding layer for any model (e.g. a SQuAD model) which takes text as input. The remaining layers of the new model are trained for the new task (e.g. SQuAD). Instructions for loading ELMo as a PyTorch module can be found here:

https://github.com/allenai/allennlp/blob/master/tutorials/how_to/elmo.md

When it was released in November 2017, ELMo improved the state of the art on six different NLP benchmarks. This established the utility of pretrained contextual embeddings, but many architectural advances have happened since November 2017. As such, ELMo is unlikely to compete with more recent PCE methods such as BERT.

5.1.2 BERT

Original paper: BERT: Pre-training of Deep Bidirectional Transformers for Language Modeling [9]

We briefly saw in class that Transformers [13] have in some cases supplanted RNNs as the dominant model family in deep learning for NLP. Therefore one might expect that ELMo's pretrained RNNs would be outperformed by pretrained Transformers. Indeed, BERT showed exactly that – by training deep Transformers on a carefully designed bidirectional language modeling task, BERT achieves state-of-the-art results on a wide variety of NLP benchmarks – including SQuAD 2.0 (as of Sep. 2019).

To enable other researchers to use BERT as an embedding module, the BERT authors released an open-source TensorFlow implementation and pretrained weights (`https://github.com/google-research/bert`). To use these weights with PyTorch, you need an op-for-op reimplementation, such as this one:

`https://github.com/huggingface/pytorch-pretrained-BERT`.

You are welcome to adapt either of these implementations (or another one) for this homework. We highly recommend that you read the BERT paper, as they give a thorough description of how to adapt BERT for various applications, including question answering.

5.2 Non-PCE Model Types

5.2.1 Character-level Embeddings

Appears in: Bidirectional Attention Flow for Machine Comprehension [3]

Character-level embeddings allow us to condition on the internal structure of words (this is known as *morphology*), and better handle out-of-vocabulary words – these are advantages compared to lookup-based word embeddings. As mentioned in Section 4.1, the original BiDAF model includes a character-level embedding layer using character-level convnets. You could alternatively try other kinds of subword modeling mentioned in lectures.

The starter code is designed to make it straightforward to add character-level word embeddings. In particular, you will notice that the `util.SQuAD` class returns character indices, and these are loaded in `train.py` and `test.py`.

5.2.2 Self-attention

Appears in: R-Net: Machine Reading Comprehension with Self-Matching Networks[4]

Self-attention is a phrase that can have slightly different meanings depending on the setting. In a RNN-based language model, self-attention often means that the hidden state $\boldsymbol{h}_t$ attends to all the previous hidden states so far $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{t-1}$. In a context where you are encoding some text length n, self-attention might mean that $\boldsymbol{h}_t$ attends to all the hidden states $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n$ (even including itself). Transformers are built on a kind of self-attention. The main idea of self-attention is that the *query* vector is from the same set as the set of *value* vectors.

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 . The R-Net paper is one of the easier ones to understand.

5.2.3 Transformers

Appears in: QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension [14]

QANet adapts ideas from the Transformer [13] and applies them to question answering, doing away with RNNs and replacing them entirely with self-attention and convolution. The main component of the QANet model is called an Encoder Block. The Encoder Block draws inspiration

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

from the Transformer: The two modules are similar in their use of positional encoding, residual connections, layer normalization, self-attention sublayers, and feed-forward sublayers. However, an Encoder Block differs from the Transformer in its use of stacked convolutional sublayers, which use depthwise-separable convolution to capture local dependencies in the input sequence. Prior to BERT, QANet had state-of-the-art performance for SQuAD 1.1.

5.2.4 Transformer-XL

Original paper: Transformer-XL: Language Modeling with Longer-Term Dependency [15]

One recent development (January 2019) is the Transformer-XL. The motivation for Transformer-XL is to allow Transformers to learn longer-term dependencies (similar to the motivation for LSTMs vs vanilla RNNs). Transformer-XL achieves this goal by using a segment-level recurrence mechanism. Note that the authors report improved performance on both long and short sequences. Thus even for question answering, where the attention mechanism can consider the entire context at once, we might expect to see performance gains over vanilla Transformers. One idea would be to adapt ideas from Transformer-XL to the QANet architecture.

5.2.5 Additional input features

Appears in: Reading Wikipedia to Answer Open-Domain Questions (aka DrQA) [16]

Although Deep Learning is able to learn end-to-end without the need for feature engineering, it turns out that using the right input features can still boost performance significantly. For example, the DrQA model significantly boosts performance on SQuAD by including some simple but useful input features (for example, a word in the SQuAD context passage is represented not only by its word vector, but is also tagged with features representing its frequency, part-of-speech tag, named entity type, etc.). If you implement a model like this, reflect on the tradeoff between feature engineering and end-to-end learning, and comment on it in your report.

5.3 More models and papers

The models and techniques we have presented here are far from exhaustive. There are many published papers on SQuAD, some of which can be found on the SQuAD leaderboard, and others via searching online – there may be new ones that we haven't seen yet! In addition, there is lots of deep learning research on question answering and reading comprehension tasks other than SQuAD . These papers may contain interesting ideas that you can apply to SQuAD.

5.4 Other improvements

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!

- **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 300-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 or fine-tuning the embeddings.

- **Combining forward and backward 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 LSTM. You could try a GRU instead – it might be faster.
- **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 Adadelta optimizer. PyTorch 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 Submitting to the Leaderboard

6.1 Overview

We are hosting four leaderboards on Gradescope, where you can compare your performance against that of your classmates. F1 score is the performance metric we will use to rank submissions, although both EM and F1 scores will be displayed. The leaderboards can be found at the following links:

1. **Pretrained Contextual Embedding (PCE) Division.**
   (a) **Dev:** see QQ group
   (b) **Test:** see QQ group
2. **Non-PCE Division.**
   (a) **Dev:** see QQ group
   (b) **Test:** see QQ group

Recall that you should choose a division based on whether you use PCE (BERT/ELMo) or not. Within your division, you may submit to the dev leaderboard as many times as you like, but **you will only be allowed 3 successful submissions to the test leaderboard**. For your final report, we will ask you to choose a single test leaderboard submission to consider for your final performance. Therefore you must make at least one submission to the test leaderboard, but be careful not to use up your test submissions before you have finished developing your best model.

Your submission is a CSV file of answers on the dev/test set. You may use the starter code's `test.py` script to generate a submission file of the correct format, or see lines 128-135 for example code to generate a submission file. At a high level, the submission file should look like the following:

```
Id,Predicted
001fefa37a13cdd53fd82f617,Governor Vaudreuil
00415cf9abb539fbb7989beba,May 1754
00a4cc38bd041e9a4c4e545ff,
...
fffcaebf1e674a54ecb3c39df,1755
```

The header is required, and each subsequent row must contain two columns: the first column is a 25-digit hexadecimal ID for the question/answer example (IDs defined in `{dev,test}-v2.0.json`), and the second column is your predicted answer (or the empty string for no answer). The rows can be in any order. For the test leaderboard, you must submit a prediction for every example, and for the dev leaderboard, you must submit predictions for at least 95% of the examples (*e.g.,* to allow for the default preprocessing in `setup.py` which throws away long examples in the dev set).

6.2 Submission Steps

Here are the concrete steps for submitting to the leaderboard:

1. Generate a submission file (*e.g.,* by running `test.py` in the starter code) for either the dev or test set. Make sure to set the `--split` flag for `test.py` accordingly.
2. Save the submission file locally under the name `{dev,test}_submission.csv`.
3. Use the URLs above to navigate to the leaderboard. **Make sure to choose the correct leaderboard for your split (DEV *vs.* TEST) and division (PCE *vs.* NON-PCE).**
4. Choose the CSV file to upload.
5. Click upload and wait for your scores. The submission output will tell you the submission EM/F1, although the leaderboard will keep scores for the submission with the highest F1 score thus far.

7 Grading Criteria

The homework 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, the effort you applied, and the quality of your write-up, evaluation, and error analysis. Generally, implementing more complicated models represents more effort, and implementing more unusual models (e.g. ones that we have not mentioned in this handout) represents more creativity. You are not required to pursue original ideas, but the best homeworks in this class will go beyond the ideas described in this handout, and may in fact become published work themselves!

There is no pre-defined F1 or EM score to ensure a good grade. Though we have run some preliminary tests to get some ballpark scores, it is impossible to say in advance what distribution of scores will be reasonably achievable for students in the provided timeframe.

For similar reasons, there is no pre-defined rule for which of the models in Section 5 (or elsewhere) would ensure a good grade. Implementing a small number of things with good results and thorough experimentation/analysis is better than implementing a large number of things that don't work, or barely work. In addition, the quality of your writeup and experimentation is important: we expect you to convincingly show that your techniques are effective and describe why they work (or the cases 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?

We will expect more complex things implemented, more thorough experimentation, and better results .

8 Honor Code

Any honor code guidelines that apply for the homework in general also apply for the default final project. Here are some guidelines that are specifically relevant to the default final project:

1. You are allowed to use whatever existing code, libraries, or data you wish, with one exception for the non-PCE division (see Rule 2). However, you must clearly cite your sources and indicate which parts of the homework 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. You will be graded based on what you add on top of others' work.
2. In the non-PCE division, you **may not** use a pre-existing implementation for the SQuAD challenge 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), send an email to get permission.
   (a) *Note:* This rule does not apply to the PCE division. You may use whatever code you like in the PCE division, provided you acknowledge it explicitly in your code and paper.
3. As described in Section 3.1, it is an honor code violation to use the official SQuAD dev set in any way.
4. 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 student's code, or incorporate their code into your homework.
5. Do not share your code publicly (e.g., in a public GitHub repo) until after the class has finished.

9 FAQs

9.1 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).

9.2 How are padding and truncation handled?

In the baseline code, the hyperparameters `para_limit` and `ques_limit` define the maximum length (in words) of the context and question, set to 400 and 50 respectively. Any truncation is performed during pre-processing. During training, we dynamically pad each batch to the maximum length occurring in that batch. 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. See the `collate_fn` function in `util.py` for details.

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.

9.3 Which parts of the code can I change?

When you submit your results to our leaderboard, you will just be uploading a CSV file of your model-produced answers. This is our only requirement of your code – that it can produce this file. Therefore you could, if you wish, delete all the provided code and write your own from scratch – as long as it produces the required CSV file. More likely, you will mostly change `models.py` and `layers.py`.

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] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

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

[4] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[7] Omer Levy, Minjoon Seo, Eunsol Choi, and Luke Zettlemoyer. Zero-shot relation extraction via reading comprehension. *arXiv preprint arXiv:1706.04115*, 2017.

[8] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[11] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[12] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[14] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. Qanet: Combining local convolution with global self-attention for reading comprehension. *arXiv preprint arXiv:1804.09541*, 2018.

[15] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Language modeling with longer-term dependency. 2018.

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

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

[18] 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.