Reading Comprehension on SQuAD

Zachary Taylor, Brexton Pham, Meghana Rao
Stanford University, Department of Computer Science
CS224n Winter 2017
Codalab username: ztaylor

Abstract

We implement three models to answer questions in the SQuAD dataset based on their context paragraphs. We compare their results. Our first model consists of a question encoding using a bidirectional LSTM, a context encoding using an LSTM and taking the question into account, a straightforward attention mechanism, and a linear decoding. Our second model is an extension of this first model to include Dynamic Coattention. Our third model is also our simplest, and consists of separate question and context encodings using LSTMs and a dot product decoding of each context hidden state with the question representation. Initial evaluations demonstrated overfitting on the training data. The team chose to focus on debugging and understanding errors with a barebones baseline model. Ultimately, the model achieved a 24.25 F1 score and a14.01 exact match score.

1 Introduction

1.1 The Task

The objective of this research is to build a question-answering machine comprehension system that performs on the SQuAD dataset. In the SQuAD task, answering a question is defined as predicting an answer span within a given context paragraph. The goal is to predict an answer span tuple {as, ae} given a question of length n, q = {q1, q2, ..., qn}, and a supporting context paragraph p = {p1, p2, ..., pm} of length m. Therefore, the model must comprehend the question and reason through the passage to identify the correct answer span. The model learns a function that, given a pair of sequences (q, p) returns a sequence of two scalar indices {as, ae} indicating the start position and end position of the answer in paragraph p, respectively. Note that in this task as ≤ ae, and 0 ≤ as, ae ≤ m. Previously, the task of machine comprehension was difficult to attempt due to datasets limited in size, but the SQuAD dataset enables researchers to now build end-to-end models. It is over double the size of other current machine comprehension datasets and all questions are hand-written by humans, requiring different forms of reasoning to answer.

To approach the task, a baseline neural network model will be implemented. The architecture of the baseline model consists of feeding the context paragraph and question into an LSTM, and for each hidden state in our context representation, computing a simple dot product against the final question state to compute the probability that any index was the start or end index.

After implementing the baseline model, the objective is to implement more complex models that are inspired from current state-of-the-art research. The first advanced model to implement involves utilizing a dynamic coattention network. The model also incorporates a bidirectional LSTMs and relies on linear decoding.

1.2 Background/Related Work

Wang et al from the IBM Watson Research Center built a Multi-Perspective Context Matching (MPCM) model to approach the task of machine comprehension which predicts answer beginnings and endings in context paragraphs. The model multiplies each word embedding vector in the paragraph by a relevancy weight computed against a question and then uses a bi-directional LSTM to encode the question and passage. The model compares the context at each point in the paragraph against the question from multiple perspectives to find the beginning and ending point of the answer (Wang et al., 2016). A team of researchers from Salesforce Research implemented a dynamic coattention network which overcomes limitations that occur in a single-pass model. In their research, they implemented co-dependent representations of both the context paragraphs and the questions and then iterate over all potential answer spans using a dynamic pointing decoder. The co-dependent representations are computed by making an affinity matrix that contains affinity scores for each document word and each question word which they then fuse with temporal information using an LSTM. Their ensemble method achieved an F1 score of 80.4% on the SQuAD dataset (Socher, 2017). Jiang and Wang from Singapore Management University tackled the machine comprehension challenge by implementing a model that relies on a match-LSTM and an answer pointer. In the match-LSTM model, two sentences are fed in, where one is the premise and the other is the hypothesis. The model checks the words of the hypothesis against the words of the premise and creates a vector representation at each token. An attention mechanism is used to obtain the weighted vector representation which is then combined with the other previously constructed representation and fed into an LSTM. The answer pointer selects a position from the input text as an output symbol instead of picking an output token from a given fixed vocabulary. They achieved a 77% F1 score on the SQuAD dataset (Jiang, 2016).

2 Approach

<u>Data Preprocessing:</u>

All data preprocessing was completed in train.py. The validation dataset was built by reading through the validation answers, spans, and context files. The training dataset was built by reading through and compiling the training answers, spans, context paragraphs. Each dataset also consisted of question and context paragraph masks and placeholders for the start and end indices of the answer. Padded token vectors were appended to the ends of the context paragraphs and questions that did not align in length in the dataset. Initialization of the model also occurs in train.py and is where previous saved versions of the model can be evaluated.

<u>Baseline Model:</u>

The baseline model was implemented in qa_model.py. In the baseline model, the questions and context are both encoded using a basic LSTM. In the encoder class, the basic LSTM cell is initialized and the outputs are created by fully dynamically unrolling the inputs using a dynamic RNN cell. The decoder class takes in a knowledge representation and outputs a probability estimation over all paragraph tokens, which helps determine which token should be the start or end of the answer spam. The knowledge representation takes in q and X, our final question hidden state and our context hidden states. The class QASystem is then initialized and is passed in the encoded question and context as well as the decoder object, embedding path and question and context masks. In the QASystem class, the placeholders for the start and end answer indices, the questions, the question masks, the context paragraphs, and the context paragraph masks are made. The probability distributions for the start and end answer are initialized.

Within QASystem, the embeddings, system, and loss are also initialized. The embeddings are initialized using the glove pretrained embeddings (we use 100 dimensional glove vectors). The loss is calculated using the softmax cross entropy loss with logits, and is the sum of the loss over the start and end placeholders. The loss is optimized using the Adam optimizer.

In the main training loop of the model, minibatches of random samples from the dataset of a preset size are assembled. Within each epoch, new minibatches are made until the entire training dataset has been covered. The inputs, and start and end answer labels are optimized. After

each epoch, the loss on the validation set is calculated. To obtain the predicted start and end indices, the answer function is called, which calls a separate decode function. In decode, the probability distribution over different positions in the paragraph are returned. In the answer evaluation portion of the model, a random subsample of the validation dataset is read into the model. The F1 and exact match scores are initialized. The data is broken into segments of the size of the batch size in order to fit the placeholders initialized with the system setup.

For each sample, the string of the predicted answer is obtained by indexing into the context paragraph and grabbing the words that correspond to the predicted span of the answer. The true answer is obtained from indexing into the trained answers file. Both the predicted and true answer are passed into the F1 and EM functions and averaged.

Advnaced Model

Our most advanced model incorporated dynamic coattention (see “Dynamic Coattention Networks for Question Answering”) with our previous structure of bidirectional LSTMs and co-question-context representations. We calculate attention over the question and context representations simultaneously by first calculating an affinity matrix between them, then we introduce some nonlinearities (we used softmax). We then calculate context summaries, taking into account each word of the context, by multiplying the original question representation concatenated with the combined question representation after the softmax by the post-softmax paragraph representation. We then do a similar set of calculations to get question summaries. We then use linear transformations with learnable weights to feed into our decoder.

3 Experiments

To improve our model, we experimented with a wide variety of different techniques and hyperparameters. We had the goal of optimizing both our F1 score and our exact match (EM) score, which represent (roughly) how many of the words we predict as being in the answer actually are in the answer, as well as how many answers we get exactly right. When we initially implemented a baseline, we saw near-zero performance in terms of EM score on our validate set. We were able to overfit a small training sample, and even to overfit the overall training sample relatively well, but we weren’t seeing performance on the validate set even after 7 epochs:

Model	**F1 Score**	**EM Score**
Bidirectional Encoding w/ Attention	3.02	0.02

Partly in the attempt to fix this problem and partly in the attempt to implement a better model, we implemented a version of attention very similar to the Dynamic Coattention mechanism outlined in the paper “Dynamic Coattention Networks for Question Answering” in our references. After debugging our evaluate script and running this network on our validation set, however, we experienced similar problems:

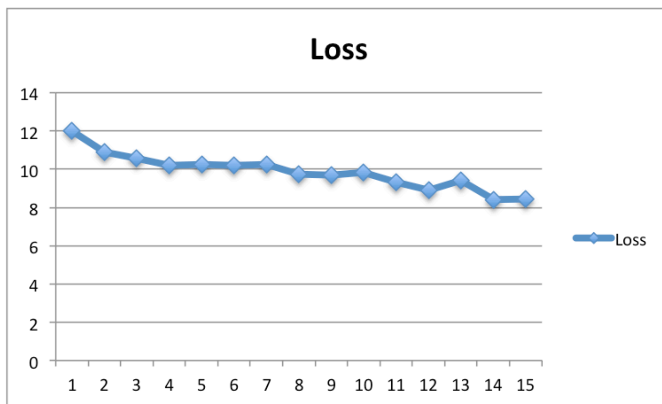
Model	**F1 Score**	**EM Score**
Dynamic Coattention	3.42	0.01

Looking deeper into these results showed that our both of our models just weren’t learning anything generalizable. For example, here are some answers that our model predicted compared

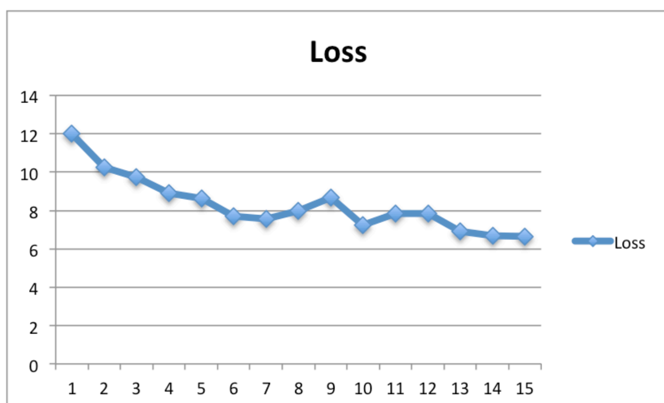
with the correct answers (there appears to be little correlation, and in fact, the lengths are consistently much longer than the correct answers):

Predicted Answer	Correct Answer
The Sultanate of Ifat , led by the Walashma dynasty with its	the Red Sea
Europe regularly for holidays . In 1889	Ireland
Highland Quichuas living in the valleys of the Sierra region . Primarily consisting	Inca
Kingdom of Great Britain and Northern Ireland and its interests and	Great Britain and Northern Ireland

As the following loss chart (loss during training) for our Dynamic Coattention model shows, we were fitting our train data at least somewhat (the loss decreases a little bit), but the test performance was not good. We suspect that there was a bug in our model somewhere.



At this point in the project, we had invested incredible amounts of time but we were still seeing no results and had limited time remaining, so we decided to experiment with an extremely simple model in order to get some limited performance. To this end, we designed our simplest baseline model that fed the question and context through separate LSTMs and computed dot products as scores for both the start and the end index. This allowed us to train epochs extremely quickly compared to before (~30 minutes vs 3 hours), which in turn allowed us to debug our model more quickly and get the framework correct at a simple level. One experimental change that ended up helping us was to decrease the learning rate. By using a learning rate that started at .001 instead of .01 (the Adam Optimizer theoretically changes this, but it still effected our learning), our simple model seemed to make more progress in terms of loss after many epochs:



Our simple model, as seen in the graph above, achieved a sustained lower loss than the other models (it levelled out at around 6.5). This wasn't sure proof that we had fixed our problem, but we were delighted to find that this model actually was doing much better than our previous ones on the validation set. It achieved the following performance on the small subset we were predicting on:

Model	**F1 Score**	**EM Score**
Separate Encodings	22.51	9.96

We then dug deeper to find out why the model was performing better. We looked at examples such as the following:

Predicted Answer	**Correct Answer**
hunting	no natural predators
claims	religious bigotry
March	Bonus March
1639	1639
Catholic	true apostolic succession
Toronto	Toronto
system	Jawed
inventor	universities

The examples above show a couple answers that our models predicts correctly. Given the time crunch, we were not able to get our more advanced models to surpass this performance, but we did tune hyperparameters of our simple model (learning rate, batch size, hidden state size, etc) in order to find a good balance of speed, which allowed for more training, and flexibility. This eventually led to our **FINAL TEST SET RESULTS (on codalab)**:

Model	**F1 Score**	**EM Score**

Separate Encodings	24.25	14.01

In the analysis section, we will discuss what the drawbacks and the advantages of our model were, and what specific types of questions it answers poorly and well.

Evaluation:

Each model was trained for 10 epochs, and each epoch consisted of 2544 minibatches. Every 500 minibatches, the model was saved at its current timestamp in the .tmp directory in order to be evaluated. The validation cost on the validation dataset was calculated after each epoch using softmax cross entropy loss with logits. To evaluate the model, the F1 score and exact match score were calculated, two metrics introduced in the original SQuAD paper. Exact match measures the percentage of the predicted answers that match the exact ground truth answers. The F1 score is the average overlap between the ground truth and prediction, and is calculated by multiplying the precision and recall scores by two and dividing by their sum. The F1 score and exact match score were calculated as the average of the scores over the number of samples evaluated.

4 Analysis

Our prediction result on dev-v1.1.json resulted in performance metrics of 23.85 and 13.556 for F1 and EM, respectively. One crucial characteristic of our model that was both a help and a hindrance to the model's performance, based on the examples in the experiments section, was its limitation to predicting one word answers. Since we used the same decoding for the start and end labels, each start and end label was the same. This meant that the model got none of the answers correct that were more than one word, which is an obvious and large drawback. However, it also didn't have the downside that our earlier models had of predicting spans that were too long in length. In fact, since there were a fair amount of one-word answers in the dataset, the model was able to get about 13% of the questions correct (13.556 EM score), and it also managed to sometimes pick a word that was part of the answer even if it didn't capture the whole thing (23.85 F1). Another interesting phenomena we noticed was that our model was better at predicting certain types of one word answers than others. For example, if the correct answer was a name or a year, it was more likely to guess that answer correctly. We hypothesize that this is because the frequency of overall answers that are proper nouns or years is higher than other types of words, and the model learned to differentiate those. Also, there are usually clear identifiers in the question that would let the model know that it is looking for a certain category of noun ('when', 'where', 'who', etc). This probably implies that our simple model is not utilizing context as well as it could, but does take into account the likelihood based on word type. For example, consider the following visualization of the question and context paragraph that led to one of the answers from before:

Visualization

Question: Where is the Center for New Religions located ?

Context Paragraph: Robert S. Wood has argued that the United States is a model for the world in terms of how a separation of church and state—no state-run or state-established church—is good for both the church and the state , allowing a variety of religions to flourish . Speaking at the Toronto-based Center for New Religions , Wood said that the freedom of conscience and assembly allowed under such a system has led to a " remarkable religiosity " in the United States that is n't present in other industrialized nations . Wood believes that the U.S. operates on " a sort of civic religion , " which includes a generally-shared belief in a creator who " expects better of us . " Beyond that , individuals are free to decide how they want to believe and fill in their own

creeds and express their conscience . He calls this approach the " genius of religious sentiment in the United States . "

Predicted Answer, Correct Answer: Toronto, Toronto

Analysis: Our suspicion is that the level that our simple model is operating at is one where it didn't necessarily take the word "based" into account, as we were not using a bidirectional LSTM, making it hard to capture that relationship because the word "based" came later in the sentence. We would only capture forward relationships. Rather, our model probably managed to get the right answer by accurately representing the key word "located" as part of the question representation, and learning that the word "located" corresponds to location answers, such as Toronto. Being the only clear city name in the paragraph, it chose it in this case, though the model would likely not be able to differentiate very well between multiple city options if they were present (we saw this is in some other examples).

5 Conclusion

After completing this project, we have gained perspective on the challenges involved with implementing a successful neural network for the SQuAD reading comprehension task. We ultimately demonstrated a functioning baseline model after investing into implementing more complex models that had long training durations and overfit the training data. Initial successes on the training data with more complex models did not perform well on the validation set and thus catalyzed a series of simplifications and debugging procedures to scale back the model to a leaner and more iterable scale. In retrospect, we wish to have implemented the barebones model first before attempting to implement more ambitious models. Nevertheless, we at least managed to get some positive performance eventually, and we learned an incredible amount about implementing neural network models for NLP. Also, we thought that even 13% of correct answers was a worthy accomplishment on such a difficult machine comprehension task.

For possible future directions, an obvious area of improvement would be changing the model to be able to predicted longer-than-one-word answers. This would involve careful experimentation and modification of the one model that we managed to get some meaningful performance out of. We could then graduate to implementing more advanced attention mechanisms. Potentially, we could learn what about our more advanced models didn't manage to translate to the evaluation set, and fix that element. In terms of other advanced models that we are interested in, answer pointers seem like a potentially valuable next step to take if we got the other elements working.

6 Contributions

Zach- Implemented the data preprocessing/reading into the model. Implemented the full baseline model, including read in, variable set up, loss, optimization, encoder/decoder, etc. Implemented the dynamic coattention model. Implemented the simple baseline model that achieved our group's top performance. Implemented qa_answer.py to produce predictions for evaluation. Went through the codalab submission process and was responsible for producing results on the leaderboard. Wrote the "advanced models", "experiments", and "analysis" portions of the paper. Contributed to the poster.

Meghana - Implemented evaluation answer, decode, and validation cost in baseline model. Monitored multiple training runs of the baseline model and worked on debugging the saving and reloading mechanism. Initialized validation dataset. Wrote the introduction, background, conclusion, and baseline approach of the paper and formatted the paper. Main person in charge of managing VM resources, etc. Formatted and contributed to the poster.

Brexton - Also implemented evaluation answer, decode, and validation cost. Contributed to

debugging the saving and reloading mechanism. Implemented methods of validating and evaluating on larger batch sizes. Spent lots of time debugging the performance of the initial models. Ran multiple tests on the virtual environment of baseline and advanced models. Contributed to poster and background information. Main person in charge of previous work research.

7 References

Chen, D., Bolton, J., Manning, C. 2016. Stanford Attentive Reader. A Thorough Examination of the CNN/Daily Mail Reading Comprehension Task. arXiv preprint arXiv:1606.02858v2

Jiang, J., Wang, S. 2016. Match-LSTM with Answer-Pointer. arXiv preprint arXiv: 1608.07905

Lee et al., 2017. Recurrent Span Representation, RaSoR. arXiv preprint arXiv:1611.01436v2.

Seo et al., 2017. Bi-directional Attention Flow, BiDAF. arXiv preprint arXiv arXiv:1611.01603v5.

Socher, R., Xiong, C., Zhong, V. 2016. Dynamic Coattention Networks for Question Answering. arXiv preprint arXiv:1611.01604

Wang et al., 2016. Multi-perspective context matching for machine comprehension. arXiv preprint arXiv:1612.04211, 2016

Yang et al., 2016. Words or Characters? Fine-Grained Grating for Reading Comprehension. arXiv preprint arXiv:1611.01724v1.

Yu et al., 2016. Dynamic Chunk Reader. arXiv preprint arXiv:1610.09996v2.

8 Supplementary Materials

train.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import json

import tensorflow as tf
import numpy as np

from qa_model import Encoder, QASystem, Decoder
from os.path import join as pjoin

import logging
import sys

logging.basicConfig(level=logging.INFO)

tf.app.flags.DEFINE_float("training_cap", 32, "Training cap.")
tf.app.flags.DEFINE_float("learning_rate", 0.001, "Learning rate.")
tf.app.flags.DEFINE_float("max_gradient_norm", 20.0, "Clip gradients
to this norm.")
tf.app.flags.DEFINE_float("dropout", 0.15, "Fraction of units randomly
dropped on non-recurrent connections.")
tf.app.flags.DEFINE_integer("batch_size", 32, "Batch size to use
during training.")
tf.app.flags.DEFINE_integer("epochs", 15, "Number of epochs to
train.")
```

```
#tf.app.flags.DEFINE_integer("sampleSize", 40, "Number of epochs to train.") #this is the number of samples fed into evaluate answer
tf.app.flags.DEFINE_integer("state_size", 124, "Size of each model layer.")
tf.app.flags.DEFINE_string("train_dir", "train", "Training directory (default: ./train).")
tf.app.flags.DEFINE_integer("output_size", 400, "The output size of your model.")
tf.app.flags.DEFINE_integer("embedding_size", 100, "Size of the pretrained vocabulary.")
tf.app.flags.DEFINE_string("data_dir", "data/squad", "SQuAD directory (default ./data/squad)")
tf.app.flags.DEFINE_string("load_train_dir", "train", "Training directory to load model parameters from to resume training (default: {train_dir}).")
tf.app.flags.DEFINE_string("log_dir", "log", "Path to store log and flag files (default: ./log)")
tf.app.flags.DEFINE_string("optimizer", "adam", "adam / sgd")
tf.app.flags.DEFINE_integer("print_every", 1, "How many iterations to do per print.")
tf.app.flags.DEFINE_integer("keep", 0, "How many checkpoints to keep, 0 indicates keep all.")
tf.app.flags.DEFINE_string("vocab_path", "data/squad/vocab.dat", "Path to vocab file (default: ./data/squad/vocab.dat)")
tf.app.flags.DEFINE_string("embed_path", "", "Path to the trimmed GLoVe embedding (default: ./data/squad/glove.trimmed.{embedding_size}.npz)")

FLAGS = tf.app.flags.FLAGS

def initialize_model(session, model, train_dir):
    print('train dir ', train_dir)
    ckpt = tf.train.get_checkpoint_state(train_dir)
    print('ckpt ', ckpt)
    v2_path = ckpt.model_checkpoint_path + ".index" if ckpt else ""
    print('v2_path ', v2_path)
    if ckpt and (tf.gfile.Exists(ckpt.model_checkpoint_path) or tf.gfile.Exists(v2_path)):
        logging.info("Reading model parameters from %s" % ckpt.model_checkpoint_path)
        print('ckpt.model_checkpoint_path ', ckpt.model_checkpoint_path)
        model.saver.restore(session, ckpt.model_checkpoint_path)
    else:
        logging.info("Created model with fresh parameters.")
        session.run(tf.global_variables_initializer())
        logging.info('Num params: %d' % sum(v.get_shape().num_elements() for v in tf.trainable_variables()))
    return model


def initialize_vocab(vocab_path):
    if tf.gfile.Exists(vocab_path):
        rev_vocab = []
        with tf.gfile.GFile(vocab_path, mode="rb") as f:
            rev_vocab.extend(f.readlines())
        rev_vocab = [line.strip('\n') for line in rev_vocab]
        vocab = dict([(x, y) for (y, x) in enumerate(rev_vocab)])
        return vocab, rev_vocab
    else:
        raise ValueError("Vocabulary file %s not found.", vocab_path)


def get_normalized_train_dir(train_dir):
    """
```

```
    Adds symlink to {train_dir} from /tmp/cs224n-squad-train to
canonicalize the
    file paths saved in the checkpoint. This allows the model to be
reloaded even
    if the location of the checkpoint files has moved, allowing usage
with CodaLab.
    This must be done on both train.py and qa_answer.py in order to
work.
    """
    #global_train_dir = './train/20170316_002002'
    #global_train_dir = './train/20170316_002002'
    # global_train_dir = './train/20170316_002002/model.weights'

    global_train_dir = "/tmp/cs224n-squad-train"
    # global_train_dir = train_dir

    # os.remove(global_train_dir)
    if os.path.exists(global_train_dir):
        os.unlink(global_train_dir)
    if not os.path.exists(train_dir):
        os.makedirs(train_dir)
    os.symlink(os.path.abspath(train_dir), global_train_dir)
    return global_train_dir


def main(_):

    embed_path = FLAGS.embed_path or pjoin("data", "squad",
"glove.trimmed.{}.npz".format(FLAGS.embedding_size))
    vocab_path = FLAGS.vocab_path or pjoin(FLAGS.data_dir,
"vocab.dat")
    vocab, rev_vocab = initialize_vocab(vocab_path)

    def padded_token_vector(vector, max_length):
        return [vocab[vector[i]] if i < len(vector) else vocab['<pad>'] for
i in range(max_length)]
    # Do what you need to load datasets from FLAGS.data_dir

    ####################################
    # TRAIN DATASET
    ####################################

    questions = []
    question_mask = []
    with open(FLAGS.data_dir + "/train.question") as file:
        for line in file:
            q = str.split(line)
            question_mask.append(len(q))
            questions.append(padded_token_vector(q, 60))


    overflow_indices = []
    context = []
    context_mask = []
    with open(FLAGS.data_dir + "/train.context") as file:
        #i = 0
        for i, line in enumerate(file):
            #if i < 32:
            c = str.split(line)
            if len(c) > FLAGS.output_size: overflow_indices.append(i)
            context_mask.append(len(c))
            context.append(padded_token_vector(c, FLAGS.output_size))
                #i += 1

    contextWords = [] #get all the words from each context
```

```
    with open(FLAGS.data_dir + "/train.context") as file:
        #i = 0
        for line in file:
            #if i < 32:
            newContext = []
            c = str.split(line)
            for eachContext in c:
                word = eachContext.split(" ")
                newContext.append(word)
            contextWords.append(newContext)
                #i += 1

    labels = []
    labels_start = []
    labels_end = []

    with open(FLAGS.data_dir + "/train.span") as file:
        lineNum = 0
        #i = 0
        for line in file:
            #if i < 32:
            start, end = [int(s) for s in str.split(line)]
            labels_start.append(start)
            labels_end.append(end)
            start_word = contextWords[lineNum][start]
            end_word = contextWords[lineNum][end]
            lineNum = lineNum + 1
                # print(start_word, end_word)
                # sys.exit()
                #create these two rarrayss)
                #i += 1

    trainAnswers = []
    with open(FLAGS.data_dir + "/train.answer") as file:
        trainAnswers = file.readlines()
        trainAnswers = [x.strip() for x in trainAnswers]#[:32]

    contextParagraphs = []
    with open(FLAGS.data_dir + "/train.context") as file:
        contextParagraphs = file.readlines()
        contextParagraphs = [x.strip() for x in contextParagraphs]#[:32]


    ###################################
    # VALIDATE DATASET
    ###################################

    val_questions = []
    val_question_mask = []
    with open(FLAGS.data_dir + "/val.question") as file:
        for line in file:
            q = str.split(line)
            val_question_mask.append(len(q))
            val_questions.append(padded_token_vector(q, 60))

    val_overflow_indices = []
    val_context = []
    val_context_mask = []
    with open(FLAGS.data_dir + "/val.context") as file:
        for i, line in enumerate(file):
            c = str.split(line)
            if len(c) > FLAGS.output_size: val_overflow_indices.append(i)
            val_context_mask.append(len(c))
            val_context.append(padded_token_vector(c,
FLAGS.output_size))
```

```
    val_contextWords = [] #get all the words from each context
    with open(FLAGS.data_dir + "/val.context") as file:
        for line in file:
            newContext = []
            c = str.split(line)
            for eachContext in c:
                word = eachContext.split(" ")
                newContext.append(word)
            val_contextWords.append(newContext)

    val_labels = []
    val_labels_start = []
    val_labels_end = []
    with open(FLAGS.data_dir + "/val.span") as file:
        lineNum = 0
        for line in file:
            start, end = [int(s) for s in str.split(line)]
            val_labels_start.append(start)
            val_labels_end.append(end)

    valAnswers = []
    with open(FLAGS.data_dir + "/val.answer") as file:
        valAnswers = file.readlines()
        valAnswers = [x.strip() for x in valAnswers]#[:32]

    valParagraphs = []
    with open(FLAGS.data_dir + "/val.context") as file:
        valParagraphs = file.readlines()
        valParagraphs = [x.strip() for x in contextParagraphs]#[:32]



    for i in reversed(overflow_indices):
        questions.pop(i)
        question_mask.pop(i)
        context.pop(i)
        context_mask.pop(i)
        contextWords.pop(i)
        labels_start.pop(i)
        labels_end.pop(i)
        trainAnswers.pop(i)
        contextParagraphs.pop(i)
    #labels = []
    #with open(FLAGS.data_dir + "/train.span") as file:
     #   for line in file:
      #     start, end = [int(s) for s in str.split(line)]

            #a = [1 if i >= start and i <= end else 0 for i in range(766)]
       #    labels_start.append(start)
        #   labels_end.append(end)
            #create these two rarrayss)

    inputs = (questions, context, question_mask, context_mask)
    val_dataset = (val_questions, val_context, val_question_mask,
val_context_mask, val_labels_start, val_labels_end, val_contextWords,
valAnswers, valParagraphs)
    # dataset = (questions, context, question_mask, context_mask,
labels)
    dataset = (questions, context, question_mask, context_mask,
labels_start, labels_end, contextWords, trainAnswers,
contextParagraphs)
```

```
    question_encoder = Encoder(size=FLAGS.state_size)
    context_encoder = Encoder(size=FLAGS.state_size)

    decoder = Decoder(output_size=FLAGS.output_size)

    qa = QASystem(question_encoder, context_encoder, decoder,
embed_path, question_mask, context_mask)

    if not os.path.exists(FLAGS.log_dir):
        os.makedirs(FLAGS.log_dir)
    file_handler = logging.FileHandler(pjoin(FLAGS.log_dir, "log.txt"))
    logging.getLogger().addHandler(file_handler)

    print(vars(FLAGS))
    with open(os.path.join(FLAGS.log_dir, "flags.json"), 'w') as fout:
        json.dump(FLAGS.__flags, fout)

    with tf.Session() as sess:
        #load_train_dir = get_normalized_train_dir(FLAGS.load_train_dir
or FLAGS.train_dir)
        # load_train_dir = '/tmp/cs224n-squad-train/20170319_160403'
        load_train_dir = './train/20170321_110723'
        initialize_model(sess, qa, load_train_dir)
        save_train_dir = get_normalized_train_dir(FLAGS.train_dir)
        qa.train(sess, dataset, save_train_dir, val_dataset)
        qa.evaluate_answer(sess, val_dataset, log=True)


if __name__ == "__main__":
    tf.app.run()
```

qa_model.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import time
import pdb
import logging
import datetime
import os
import random

import sys
import numpy as np
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf
from tensorflow.python.ops import variable_scope as vs

from evaluate import exact_match_score, f1_score

from tensorflow.python.ops.nn import
```

```
sparse_softmax_cross_entropy_with_logits as ssce

logging.basicConfig(level=logging.INFO)

def get_optimizer(opt):
   if opt == "adam":
      optfn = tf.train.AdamOptimizer
   elif opt == "sgd":
      optfn = tf.train.GradientDescentOptimizer
   else:
      assert (False)
   return optfn

class Encoder(object):
   def __init__(self, size):
      self.size = size
      self.FLAGS = tf.app.flags.FLAGS

   def encode(self, inputs, masks, scope_name,
encoder_state_input=None):
      """
      In a generalized encode function, you pass in your inputs,
      masks, and an initial
      hidden state input into this function.

      :param inputs: Symbolic representations of your input
      :param masks: this is to make sure tf.nn.dynamic_rnn doesn't
iterate
                 through masked steps
      :param encoder_state_input: (Optional) pass this as initial
hidden state
                              to tf.nn.dynamic_rnn to build conditional
representations
      :return: an encoded representation of your input.
              It can be context-level representation, word-level
representation,
              or both.
      """
      with vs.variable_scope(scope_name):
         cell = tf.nn.rnn_cell.BasicLSTMCell(self.size)
         if encoder_state_input == None:
            initial_state =  cell.zero_state(self.FLAGS.batch_size,
tf.float64)
            # outputs, (output_fw, output_bw) =
tf.nn.bidirectional_dynamic_rnn(cell_fw=cell, cell_bw=cell,
inputs=inputs, sequence_length=masks, initial_state_fw=initial_state,
initial_state_bw=initial_state)
            # c, h = (tf.concat(1, [output_fw.c, output_bw.c]),
tf.concat(1, [output_fw.h, output_bw.h]))
            # output = tf.nn.rnn_cell.LSTMStateTuple(c, h)
            # output_states = tf.concat(2, [outputs[0], outputs[1]])
            output_states, output = tf.nn.dynamic_rnn(cell,
inputs=inputs, sequence_length=masks, initial_state=initial_state)
            # output = tf.concat(1, [output.c, output.h])



         else:
            initial_state = encoder_state_input
            output_states, output = tf.nn.dynamic_rnn(cell,
inputs=inputs, sequence_length=masks, initial_state=initial_state)

         return output.h, output_states
```

```
class Decoder(object):
    def __init__(self, output_size):
        self.output_size = output_size
        self.FLAGS = tf.app.flags.FLAGS

    def decode(self, knowledge_rep):
        """
        takes in a knowledge representation
        and output a probability estimation over
        all paragraph tokens on which token should be
        the start of the answer span, and which should be
        the end of the answer span.

        :param knowledge_rep: it is a representation of the paragraph
and question,
                              decided by how you choose to implement the
encoder
        :return:
        """
        boundary_model = True
        if boundary_model:
            # h_q, h_p = knowledge_rep
            # with vs.variable_scope("answer_start"):
            #     a_s = tf.nn.rnn_cell._linear([h_q, h_p], self.output_size,
True, 1.0)
            # with vs.variable_scope("answer_end"):
            #     a_e = tf.nn.rnn_cell._linear([h_q, h_p], self.output_size,
True, 1.0)
            # return (a_s, a_e)
            q, X = knowledge_rep
            a = tf.squeeze(tf.matmul(X, tf.reshape(q, [-1,
self.FLAGS.state_size, 1])))
            return (a, a)

        else:
            P = knowledge_rep
            print('P: ', P)
            h_size = self.FLAGS.state_size
            W_s = tf.get_variable("W_s", shape =[1, h_size, 1], initializer
= tf.contrib.layers.xavier_initializer(), dtype=tf.float64)
            tiled_W_s = tf.tile(W_s, tf.pack([self.FLAGS.batch_size, 1, 1]))
            S = tf.squeeze(tf.matmul(P, tiled_W_s))
            print('S: ', S)
            cell = tf.nn.rnn_cell.BasicLSTMCell(h_size)
            initial_state = cell.zero_state(self.FLAGS.batch_size,
tf.float64)
            P_new, final_states = tf.nn.dynamic_rnn(cell, inputs=P,
initial_state=initial_state)
            print('final LSTM outputs', P_new)
            W_e = tf.get_variable("W_e", shape =[1, h_size, 1], initializer
= tf.contrib.layers.xavier_initializer(), dtype=tf.float64)
            tiled_W_e = tf.tile(W_e, tf.pack([self.FLAGS.batch_size, 1, 1]))
            E = tf.squeeze(tf.matmul(P_new, tiled_W_e))
            print('E', E)
            return S, E


class QASystem(object):
    def __init__(self, question_encoder, context_encoder, decoder,
*args):
        """
        Initializes your System
        :param encoder: an encoder that you constructed in train.py
        :param decoder: a decoder that you constructed in train.py
        :param args: pass in more arguments as needed
```

```
        """
        self.FLAGS = tf.app.flags.FLAGS

        self.question_encoder = question_encoder
        self.context_encoder = context_encoder
        self.decoder = decoder
        self.max_question_length = 60 #ran script to figure this out
        self.max_context_length = self.FLAGS.output_size

        self.embed_path = args[0]
        self.question_mask = args[1]
        self.context_mask = args[2]
        # ==== set up placeholder tokens ========
        self.labels_placeholder = tf.placeholder(tf.int32, shape=[None,
self.max_context_length])
        self.labels_start_placeholder = tf.placeholder(tf.int32,
shape=[self.FLAGS.batch_size])
        self.labels_end_placeholder = tf.placeholder(tf.int32,
shape=[self.FLAGS.batch_size])

        self.questions_placeholder = tf.placeholder(tf.int32,
shape=[None, self.max_question_length])
        self.question_mask_placeholder = tf.placeholder(tf.int32,
shape=[None,])
        self.context_placeholder = tf.placeholder(tf.int32, shape=[None,
self.max_context_length])
        self.context_mask_placeholder = tf.placeholder(tf.int32,
shape=[None,])

        self.a_s = None
        self.a_e = None

        self.contextWords = None
        #use simpler form for now\\ self.attention = tf.get_variable("A",
shape=[], initializer=tf.contrib.layers.xavier_initializer())

        # ==== assemble pieces ====
        with tf.variable_scope("qa",
initializer=tf.uniform_unit_scaling_initializer(1.0)):
            self.setup_embeddings(self.embed_path)
            self.setup_system()
            self.setup_loss()

        # ==== set up training/updating procedure ====
        optfn = get_optimizer("adam")
        optimizer = optfn(self.FLAGS.learning_rate)

        grads_and_vars = optimizer.compute_gradients(self.loss)
        gradients, parameters = zip(*grads_and_vars)
        gradients = tf.clip_by_global_norm(gradients,
self.FLAGS.max_gradient_norm)[0]
        self.grad_norm = tf.global_norm(gradients)
        new_grads_and_vars = zip(gradients, parameters)
        self.train_op = optimizer.apply_gradients(new_grads_and_vars)

        pass


    def setup_system(self):
        """
        After your modularized implementation of encoder and decoder
        you should call various functions inside encoder, decoder here
        to assemble your reading comprehension system!
        :return:
        """
```

```
        question_rep, question_states =
self.question_encoder.encode(self.question_embeddings,
self.question_mask_placeholder, "encode_question")
        print('question rep', question_rep)
        #32 by 400

        context_rep, context_states =
self.context_encoder.encode(self.context_embeddings,
self.context_mask_placeholder, "encode_context")
        print('context states', context_states)
       #  print('context', context_rep)
       #  #context rep is 32 x 400
       #  print('q', question_rep)

       #  reshaped_question_rep = tf.reshape(question_rep, shape=[2, -1,
1, self.FLAGS.state_size])
       # # print('reshaped question', reshaped_question_rep)

       #  attention_vec =
tf.nn.softmax(tf.reduce_sum(tf.multiply(context_rep, question_rep),
axis=[3]))
       #  answer_vecs = tf.multiply(context_states,
tf.reshape(attention_vec, shape=[2, -1, self.max_context_length, 1]))
       #  #answer_vecs = tf.add_n(context_states,
tf.reshape(attention_vec, shape=[2, -1, self.max_context_length, 1]))
       #  self.prob_distribution =
self.decoder.decode(tf.reshape(answer_vecs[1,:,:,:], shape=[-1,
self.max_context_length, self.FLAGS.state_size]))
        print('context states', context_states)
        print('context', context_rep)
        #context rep is 32 x 400
        print('q', question_rep)
        boundary_model = True
        if boundary_model:

            #print('reshaped question', reshaped_question_rep)
            #attention_vec =
tf.nn.softmax(tf.reduce_sum(tf.multiply(context_rep, question_rep),
axis=[3]))
            #answer_vecs = tf.multiply(context_states,
tf.reshape(attention_vec, shape=[2, -1, self.max_context_length, 1]))
            #answer_vecs = tf.add_n(context_states,
tf.reshape(attention_vec, shape=[2, -1, self.max_context_length, 1]))
            #self.prob_distribution =
self.decoder.decode(tf.reshape(answer_vecs[1,:,:,:], shape=[-1,
self.max_context_length, self.FLAGS.state_size]))

            # reshaped_question_rep = tf.reshape(question_rep, shape=[2,
-1, 1, self.FLAGS.state_size])
            # self.a_s, self.a_e = self.decoder.decode([question_rep.h,
context_rep])
            self.a_s, self.a_e = self.decoder.decode([question_rep,
context_states])

        else:
            Q = question_states
            D = context_states
            L = tf.matmul(D, tf.transpose(Q, [0, 2, 1]))
            AD = tf.nn.softmax(L)
            AQ = tf.nn.softmax(tf.transpose(L, [0, 2, 1]))
            CQ = tf.matmul(AQ, D)
            print('Q:', Q)
            print('CQ: ', CQ)
            print('AD: ', AD)
            CD = tf.matmul(AD, tf.concat(2, [Q,CQ]))
```

```
            h_size = self.FLAGS.state_size
            W = tf.get_variable("W", shape =[1, h_size*6, h_size],
initializer = tf.contrib.layers.xavier_initializer(), dtype=tf.float64)
            b = tf.get_variable("b", shape =[1, self.max_context_length,
h_size], initializer = tf.contrib.layers.xavier_initializer(),
dtype=tf.float64)
            print('CD: ', CD)
            print('D:', D)
            tiled_W = tf.tile(W, tf.pack([self.FLAGS.batch_size, 1, 1]))
            print('tiled_W', tiled_W)
            D = tf.matmul(tf.concat(2, [CD, D]), tiled_W) + b
            print('new D: ', D)
            self.a_s, self.a_e = self.decoder.decode(D)


    def setup_loss(self):
        """
        Set up your loss computation here
        :return:
        """
        with vs.variable_scope("loss"):

            l1 = ssce(labels=self.labels_start_placeholder,
logits=self.a_s)
            l2 = ssce(labels=self.labels_end_placeholder, logits=self.a_e)
            self.loss = l1 + l2

        self.saver = tf.train.Saver()


    def setup_embeddings(self, embed_path):
        """
        Loads distributed word representations based on placeholder
tokens
        :return:
        """
        with vs.variable_scope("embeddings"):
            embedding_dict = np.load(embed_path)
            pretrained_embedding = tf.constant(embedding_dict['glove'])
            print(self.questions_placeholder)
            self.question_embeddings =
tf.reshape(tf.nn.embedding_lookup(pretrained_embedding,
self.questions_placeholder), shape=[-1, self.max_question_length,
self.FLAGS.embedding_size])
            self.context_embeddings =
tf.reshape(tf.nn.embedding_lookup(pretrained_embedding,
self.context_placeholder), shape=[-1, self.max_context_length,
self.FLAGS.embedding_size])
            print(self.question_embeddings)


    def optimize(self, session, train_x, train_y1, train_y2):
        """
        Takes in actual data to optimize your model
        This method is equivalent to a step() function
        :return:
        """
        #print('trainy1', train_y1)
        #print('trainy2', train_y2)
        input_feed = {}

        train_questions, train_context, train_q_mask, train_c_mask =
train_x

        input_feed[self.questions_placeholder] = train_questions
```

```
        input_feed[self.context_placeholder] = train_context
        input_feed[self.question_mask_placeholder] = train_q_mask
        input_feed[self.context_mask_placeholder] = train_c_mask
        input_feed[self.labels_start_placeholder] = train_y1
        input_feed[self.labels_end_placeholder] = train_y2

        output_feed = [self.train_op, self.loss]

        outputs = session.run(output_feed, input_feed) #returns all three
output feed values
        print('loss: ', np.mean(outputs[1]))
        return outputs[0]


    def decode(self, session, test_x):
        """
        Returns the probability distribution over different positions in
the paragraph
        so that other methods like self.answer() will be able to work
properly
        :return:

        """
        input_feed = {}

        question, context, question_mask, context_mask, labels_start,
labels_end, contextWord, trainAnswer, contextParagraph = zip(*test_x)

        input_feed[self.questions_placeholder] = question
        input_feed[self.context_placeholder] = context
        input_feed[self.question_mask_placeholder] = question_mask
        input_feed[self.context_mask_placeholder] = context_mask
        input_feed[self.labels_start_placeholder] = labels_start
        input_feed[self.labels_end_placeholder] = labels_end

        output_feed = [self.a_s, self.a_e]

        outputs = session.run(output_feed, input_feed)

        return outputs

    def answer(self, session, test_x):

        yp, yp2 = self.decode(session, test_x)

        a_s = np.argmax(yp, axis=1)
        a_e = np.argmax(yp2, axis=1)

        return (a_s, a_e)


    def startValidate(self, session, val_dataset):
        sample = 32
        #currently only looking at 32 examples, could be looking at
entire dataset
        val_questions, val_context, val_question_mask,
val_context_mask, val_labels_start, val_labels_end, val_contextWords,
valAnswers, valParagraphs = val_dataset
        val_things = zip(val_questions, val_context, val_question_mask,
val_context_mask, val_labels_start, val_labels_end)
        indices = [random.randint(0, len(val_questions)-1) for i in
range(sample)]
        sampleData = [val_things[i] for i in indices]
        #calculate the validation cost for the val_dataset
        #do we do this to 100 samples or the whole validation dataset3
```

```
        val_cost = 0.0
        val_cost = self.validate(session, sampleData)
        print('this is val_cost ', val_cost)
        return val_cost

    def validate(self, sess, val_things):
        """
        Iterate through the validation dataset and determine what
        the validation cost is.

        This method calls self.test() which explicitly calculates
validation cost.

        How you implement this function is dependent on how you design
        your data iteration function

        :return:
        """
        valid_cost = 0
        valid_cost = self.test(sess, val_things)
        valid_cost = (np.sum(valid_cost)*1.0)/len(val_things)
        return valid_cost

    def test(self, session, val_things):
        """
        in here you should compute a cost for your validation set
        and tune your hyperparameters according to the validation set
performance
        :return:
        """
        val_questions, val_context, val_question_mask,
val_context_mask, val_labels_start, val_labels_end = zip(*val_things)
        input_feed = {}

        input_feed[self.questions_placeholder] = val_questions
        input_feed[self.context_placeholder] = val_context
        input_feed[self.question_mask_placeholder] = val_question_mask
        input_feed[self.context_mask_placeholder] = val_context_mask
        input_feed[self.labels_start_placeholder] = val_labels_start
        input_feed[self.labels_end_placeholder] = val_labels_end

        output_feed = [self.loss]

        outputs = session.run(output_feed, input_feed)

        return outputs

    def chunks(self, l, n):
        """Yield successive n-sized chunks from l."""
        for i in range(0, len(l), n):
            yield l[i:i + n]

    def clip(self, l, size):
        """Clips list to eliminate excess data"""
        if len(l[-1]) != size:
            return l[:-1]
        else:
            return l

    def augment(self, l, size):
        if len(l[-1]) != size:
            augmented_last_batch = [l[-1][i] if i < len(l[-1]) else None for
i in range(size)]
            l.pop(-1)
            l.append(augmented_last_batch)
```

```
        return l

    def predict_answers(self, session, dataset):
        questions, context, question_mask, context_mask = dataset
        zippedData = self.chunks(zip(questions, context, question_mask,
context_mask), self.FLAGS.batch_size)
        zippedData = self.clip([item for item in zippedData],
self.FLAGS.batch_size)
        a_s = []
        a_e = []
        for chunk in zippedData:
            input_feed = {}
            question, context, question_mask, context_mask =
zip(*chunk)
            input_feed[self.questions_placeholder] = question
            input_feed[self.context_placeholder] = context
            input_feed[self.question_mask_placeholder] = question_mask
            input_feed[self.context_mask_placeholder] = context_mask
            output_feed = [self.a_s, self.a_e]
            a_s_chunk, a_e_chunk = session.run(output_feed, input_feed)
            a_s_chunk = np.argmax(a_s_chunk, axis=1)
            a_e_chunk = np.argmax(a_e_chunk, axis=1)
            a_s_chunk = a_s_chunk.tolist()
            a_e_chunk = a_e_chunk.tolist()
            for i in range(self.FLAGS.batch_size):
                a_s.append(a_s_chunk[i])
                a_e.append(a_e_chunk[i])

        #print('a_s', a_s)
        #print('a_e', a_e)
        return a_s, a_e


    def evaluate_answer(self, session, dataset, sample=32, log=False):
        #need to get this to work when sample size is not batch size
        """
        Evaluate the model's performance using the harmonic mean of F1
and Exact Match (EM)
        with the set of true answer labels

        This step actually takes quite some time. So we can only sample
100 examples
        from either training or testing set.

        :param session: session should always be centrally managed in
train.py
        :param dataset: a representation of our data, in some
implementations, you can
                        pass in multiple components (arguments) of one
dataset to this function
        :param sample: how many examples in dataset we look at
        :param log: whether we print to std out stream
        :return:
        """
        sample = 200

        questions, context, question_mask, context_mask, labels_start,
labels_end, contextWords, trainAnswers, contextParagraphs = dataset
        indices = [random.randint(0, len(questions)-1) for i in
range(sample)]
        zippedData = zip(questions, context, question_mask,
context_mask, labels_start, labels_end, contextWords, trainAnswers,
contextParagraphs)
        # sampleData = [zippedData[i] for i in indices]
        zippedData = [zippedData[i] for i in indices]#[:32]
```

```
    f1 = 0
    em = 0

        #print('zippedData', zippedData)

    zippedData = self.chunks(zippedData, self.FLAGS.batch_size)
    zippedData = [item for item in zippedData]

    zippedData = self.clip(zippedData, self.FLAGS.batch_size)
    a_s = []
    a_e = []

     #print('zipped lengths' , [len(x) for x in zippedData])
    for chunk in zippedData:
        a_s_chunk, a_e_chunk = self.answer(session, chunk)
        a_s_chunk = a_s_chunk.tolist()
        a_e_chunk = a_e_chunk.tolist()
        a_s.append(a_s_chunk)
        a_e.append(a_e_chunk)

    #print('a_s lens', [len(x) for x in a_s])
     a_s = self.clip(a_s, self.FLAGS.batch_size)
    a_e = self.clip(a_e, self.FLAGS.batch_size)
     #print('a_s lens',[len(x) for x in a_s])

    indices = self.chunks(indices, self.FLAGS.batch_size)
    indices = [item for item in indices]
    indices = self.clip(indices, self.FLAGS.batch_size)

    contextWordsCounter = 0
    #for i in range(len(zippedData)):
    for i, k in enumerate(indices):
        for j, index in enumerate(k):
            current_zippedData_bucket = zippedData[i]
            #print('czb', current_zippedData_bucket)
             current_a_s_bucket = a_s[i]
            current_a_e_bucket = a_e[i]
            #for j in range(len(current_zippedData_bucket)):
            p = contextWords[index]
            p = [item[0] for item in p]
            startAnswerIndex = current_a_s_bucket[j]
            endAnswerIndex = current_a_e_bucket[j]
            answer = (p[startAnswerIndex:endAnswerIndex + 1])
            answer =" ".join(str(x) for x in answer)
            trueAnswer = trainAnswers[index]
            print('true answer: ', trueAnswer)
            print('predicted answer: ', answer)
            f1 += f1_score(answer, trueAnswer)
            em += exact_match_score(answer, trueAnswer)
            contextWordsCounter += 1

     length = sum([len(i) for i in indices])
    f1 = (f1 * 1.0)/(length)
    em = (em * 1.0)/(length)

    print('f1 ', f1)
    print('em ', em)

    return f1, em


def train(self, session, dataset, train_dir, val_dataset):
    """
```

```
        Implement main training loop

        TIPS:
        You should also implement learning rate annealing (look into
tf.train.exponential_decay)
        Considering the long time to train, you should save your model
per epoch.

        More ambitious appoarch can include implement early stopping,
or reload
        previous models if they have higher performance than the
current one

        As suggested in the document, you should evaluate your training
progress by
        printing out information every fixed number of iterations.

        We recommend you evaluate your model performance on F1 and
EM instead of just
        looking at the cost.

        :param session: it should be passed in from train.py
        :param dataset: a representation of our data, in some
implementations, you can
                    pass in multiple components (arguments) of one
dataset to this function
        :param train_dir: path to the directory where you should save
the model checkpoint
        :return:
        """

        # some free code to print out number of parameters in your
model
        # it's always good to check!
        # you will also want to save your model parameters in train_dir
        # so that you can use your trained model to make predictions, or
        # even continue training
        def get_minibatches(data, minibatch_size, shuffle=True):
            """
            Iterates through the provided data one minibatch at at time.
You can use this function to
            iterate through data in minibatches as follows:

                for inputs_minibatch in get_minibatches(inputs,
minibatch_size):
                    ...

            Or with multiple data sources:

                for inputs_minibatch, labels_minibatch in
get_minibatches([inputs, labels], minibatch_size):
                    ...

            Args:
                data: there are two possible values:
                    - a list or numpy array
                    - a list where each element is either a list or numpy
array
                minibatch_size: the maximum number of items in a
minibatch
                shuffle: whether to randomize the order of returned data
            Returns:
                minibatches: the return value depends on data:
                    - If data is a list/array it yields the next minibatch of
data.
```

```
                - If data a list of lists/arrays it returns the next
minibatch of each element in the
                  list. This can be used to iterate through multiple data
sources
                  (e.g., features and labels) at the same time.

        """
        list_data = type(data) is list and (type(data[0]) is list or
type(data[0]) is np.ndarray)
        data_size = len(data[0]) if list_data else len(data)
        indices = np.arange(data_size)
        if shuffle:
            np.random.shuffle(indices)
        for minibatch_start in np.arange(0, data_size,
minibatch_size):
            minibatch_indices =
indices[minibatch_start:minibatch_start + minibatch_size]
            yield [minibatch(d, minibatch_indices) for d in data] if
list_data \
                else minibatch(data, minibatch_indices)

    def minibatch(data, minibatch_idx):
        if type(data) is np.ndarray:
            return data[minibatch_idx]
        else:
            result = []
            for i in minibatch_idx:
                result.append(data[i])
            return result

    tic = time.time()
    params = tf.trainable_variables()
    num_params = sum(map(lambda t:
np.prod(tf.shape(t.value()).eval()), params))
    toc = time.time()
    logging.info("Number of params: %d (retreival took %f secs)" %
(num_params, toc - tic))

    questions, context, question_mask, context_mask, labels_start,
labels_end, contextWords, trainAnswers, contextParagraphs = dataset
    self.contextWords = contextWords

    for epoch in range(self.FLAGS.epochs):

        print ("Epoch %d out of %d" % (epoch + 1,
self.FLAGS.epochs))
        i = 1
        print(self.FLAGS.batch_size)
        for q_mb, c_mb, q_m_mb, c_m_mb, labels_start_minibatch,
labels_end_minibatch in get_minibatches([questions, context,
question_mask, context_mask, labels_start, labels_end],
self.FLAGS.batch_size):
            if (len(context) - (i-1)*(self.FLAGS.batch_size)) >=
self.FLAGS.batch_size:
                if i % 10 == 0: print('Minibatch number: ', i)
                i += 1
                inputs_minibatch = (q_mb, c_mb, q_m_mb, c_m_mb)
                #print(inputs_minibatch)
                self.optimize(session, inputs_minibatch,
labels_start_minibatch, labels_end_minibatch)
                if i%500 ==0:
                    results_path =
"/{:%Y%m%d_%H%M%S}/".format(datetime.datetime.now())
                    # results_path = "/newFile"
                    # model_path = results_path + "model.weights/"
```

```
                    # global_train_dir = "/tmp/cs224n-squad-train"

                    model_path = train_dir + results_path
                    print("train_dir ", train_dir)
                    if not os.path.exists(model_path):
                        os.makedirs(model_path)
                    logging.getLogger().info("New best score! Saving
model in %s", model_path)
                    # self.saver.save(session, model_path)
                    self.saver.save(session, model_path + "model.ckpt")

          results_path =
"/{:%Y%m%d_%H%M%S}/".format(datetime.datetime.now())
          # results_path = "/newFile"
          # model_path = results_path + "model.weights/"
          model_path = train_dir + results_path
          if not os.path.exists(model_path):
              os.makedirs(model_path)
          logging.getLogger().info("New best score! Saving model in
%s", model_path)
          # self.saver.save(session, model_path)
          print("THIS IS HERE ", model_path + "/model.ckpt")
          self.saver.save(session, model_path + "model.ckpt")

          self.startValidate(session, val_dataset)
```

qa_answer.py

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import io
import os
import json
import sys
import random
from os.path import join as pjoin

from tqdm import tqdm
import numpy as np
from six.moves import xrange
import tensorflow as tf
```

```
from qa_model import Encoder, QASystem, Decoder
from preprocessing.squad_preprocess import data_from_json, maybe_download, squad_base_url, \
    invert_map, tokenize, token_idx_map
import qa_data

import logging

logging.basicConfig(level=logging.INFO)

FLAGS = tf.app.flags.FLAGS

tf.app.flags.DEFINE_float("learning_rate", 0.001, "Learning rate.")
tf.app.flags.DEFINE_float("dropout", 0.15, "Fraction of units randomly dropped on non-recurrent connections.")
tf.app.flags.DEFINE_integer("batch_size", 32, "Batch size to use during training.")
tf.app.flags.DEFINE_integer("epochs", 0, "Number of epochs to train.")
tf.app.flags.DEFINE_integer("state_size", 124, "Size of each model layer.")
tf.app.flags.DEFINE_integer("embedding_size", 100, "Size of the pretrained vocabulary.")
tf.app.flags.DEFINE_integer("output_size", 400, "The output size of your model.")
tf.app.flags.DEFINE_integer("keep", 0, "How many checkpoints to keep, 0 indicates keep all.")
#tf.app.flags.DEFINE_string("train_dir", "train/20170316_002002/model.weights", "Training directory (default: ./train).")
# tf.app.flags.DEFINE_string("train_dir", "train", "Training directory (default: ./train).")
#tf.app.flags.DEFINE_string("train_dir", "train/20170318_192824/model.weights", "Training directory (default: ./train).")
#tf.app.flags.DEFINE_string("train_dir", "train/20170319_154527/model.weights", "Training directory (default: ./train).")
tf.app.flags.DEFINE_string("log_dir", "log", "Path to store log and flag files (default: ./log)")
tf.app.flags.DEFINE_string("vocab_path", "data/squad/vocab.dat", "Path to vocab file (default: ./data/squad/vocab.dat)")
tf.app.flags.DEFINE_string("embed_path", "", "Path to the trimmed GLoVe embedding (default: ./data/squad/glove.trimmed.{embedding_size}.npz)")
tf.app.flags.DEFINE_string("dev_path", "data/squad/dev-v1.1.json", "Path to the JSON dev set to evaluate against (default: ./data/squad/dev-v1.1.json)")
tf.app.flags.DEFINE_float("max_gradient_norm", 20.0, "Clip gradients to this norm.")
tf.app.flags.DEFINE_string("optimizer", "adam", "adam / sgd")

def initialize_model(session, model, train_dir):
    ckpt = tf.train.get_checkpoint_state(train_dir)
    v2_path = ckpt.model_checkpoint_path + ".index" if ckpt else ""
    #print('checkpoint path', ckpt.model_checkpoint_path)
    #print('first bool', tf.gfile.Exists(v2_path))
```

```
    #print('second bool', tf.gfile.Exists(ckpt.model_checkpoint_path))
    if ckpt and (tf.gfile.Exists(ckpt.model_checkpoint_path) or tf.gfile.Exists(v2_path)):
        logging.info("Reading model parameters from %s" % ckpt.model_checkpoint_path)
        model.saver.restore(session, ckpt.model_checkpoint_path)
    else:
        logging.info("Created model with fresh parameters.")
        session.run(tf.global_variables_initializer())
        logging.info('Num params: %d' % sum(v.get_shape().num_elements() for v in tf.trainable_variables()))
    return model



def initialize_vocab(vocab_path):
    if tf.gfile.Exists(vocab_path):
        rev_vocab = []
        with tf.gfile.GFile(vocab_path, mode="rb") as f:
            rev_vocab.extend(f.readlines())
        rev_vocab = [line.strip('\n') for line in rev_vocab]
        vocab = dict([(x, y) for (y, x) in enumerate(rev_vocab)])
        return vocab, rev_vocab
    else:
        raise ValueError("Vocabulary file %s not found.", vocab_path)



def read_dataset(dataset, tier, vocab):
    """Reads the dataset, extracts context, question, answer,
    and answer pointer in their own file. Returns the number
    of questions and answers processed for the dataset"""

    context_data = []
    query_data = []
    question_uuid_data = []

    for articles_id in tqdm(range(len(dataset['data'])), desc="Preprocessing {}".format(tier)):
        article_paragraphs = dataset['data'][articles_id]['paragraphs']
        for pid in range(len(article_paragraphs)):
            context = article_paragraphs[pid]['context']
            # The following replacements are suggested in the paper
            # BidAF (Seo et al., 2016)
            context = context.replace("''", '" ')
            context = context.replace("``", '" ')
```

```
        context_tokens = tokenize(context)

        qas = article_paragraphs[pid]['qas']
        for qid in range(len(qas)):
            question = qas[qid]['question']
            question_tokens = tokenize(question)
            question_uuid = qas[qid]['id']

            context_ids = [str(vocab.get(w, qa_data.UNK_ID)) for w in context_tokens]
            qustion_ids = [str(vocab.get(w, qa_data.UNK_ID)) for w in
question_tokens]

            context_data.append(' '.join(context_ids))
            query_data.append(' '.join(qustion_ids))
            question_uuid_data.append(question_uuid)

    return context_data, query_data, question_uuid_data


def prepare_dev(prefix, dev_filename, vocab):
    # Don't check file size, since we could be using other datasets
    dev_dataset = maybe_download(squad_base_url, dev_filename, prefix)

    dev_data = data_from_json(os.path.join(prefix, dev_filename))
    context_data, question_data, question_uuid_data = read_dataset(dev_data, 'dev',
vocab)

    return context_data, question_data, question_uuid_data


def generate_answers(sess, model, dataset, rev_vocab):
    """
    Loop over the dev or test dataset and generate answer.

    Note: output format must be answers[uuid] = "real answer"
    You must provide a string of words instead of just a list, or start and end index

    In main() function we are dumping onto a JSON file

    evaluate.py will take the output JSON along with the original JSON file
    and output a F1 and EM
```

```
    You must implement this function in order to submit to Leaderboard.

    :param sess: active TF session
    :param model: a built QASystem model
    :param rev_vocab: this is a list of vocabulary that maps index to actual words
    :return:
    """
    question_data, context_data, question_masks, context_masks, question_uuid_data = dataset
    predict_dataset = (question_data, context_data, question_masks, context_masks)
    a_s, a_e = model.predict_answers(sess, predict_dataset)
    answers = (context_data[i][s:e + 1] for i, (s, e) in enumerate(zip(a_s, a_e)))
    answers = [" ".join(rev_vocab[x] for x in answer) for answer in answers]
    answers = [answers[i] if i < len(answers) else "overflow" for i in range(len(question_data))]
    answers_dict = {}
    for a, uuid in zip(answers, question_uuid_data):
        answers_dict[uuid] = a

    return answers_dict


def get_normalized_train_dir(train_dir):
    """
    Adds symlink to {train_dir} from /tmp/cs224n-squad-train to canonicalize the
    file paths saved in the checkpoint. This allows the model to be reloaded even
    if the location of the checkpoint files has moved, allowing usage with CodaLab.
    This must be done on both train.py and qa_answer.py in order to work.
    """
    global_train_dir = '/tmp/cs224n-squad-train'
    # global_train_dir = './train/20170316_002002/model.weights'
    if os.path.lexists(global_train_dir):
        print('lexists')
        os.unlink(global_train_dir)
    if not os.path.exists(train_dir):
        print('path does not exist')
        os.makedirs(train_dir)
    print('path: ', os.path.abspath(train_dir))
    print('global train dir: ', global_train_dir)
    os.symlink(os.path.abspath(train_dir), global_train_dir)
    return global_train_dir
```

```
def main(_):

    def padded_vector(vector, max_length):
        return [vector[i] if i < len(vector) else vocab['<pad>'] for i in range(max_length)]

    vocab, rev_vocab = initialize_vocab(FLAGS.vocab_path)

    embed_path = FLAGS.embed_path or pjoin("data", "squad", "glove.trimmed.{}.npz".format(FLAGS.embedding_size))

    if not os.path.exists(FLAGS.log_dir):
        os.makedirs(FLAGS.log_dir)
    file_handler = logging.FileHandler(pjoin(FLAGS.log_dir, "log.txt"))
    logging.getLogger().addHandler(file_handler)

    print(vars(FLAGS))
    with open(os.path.join(FLAGS.log_dir, "flags.json"), 'w') as fout:
        json.dump(FLAGS.__flags, fout)

    # ========= Load Dataset =========
    # You can change this code to load dataset in your own way
    max_question_length = 60
    max_context_length = FLAGS.output_size

    dev_dirname = os.path.dirname(os.path.abspath(FLAGS.dev_path))
    dev_filename = os.path.basename(FLAGS.dev_path)
    context_data, question_data, question_uuid_data = prepare_dev(dev_dirname, dev_filename, vocab)
    context_data = [[int(n) for n in c.split()] for c in context_data]
    question_data = [[int(n) for n in q.split()] for q in question_data]
    context_masks = [len(c) for c in context_data]
    question_masks = [len(q) for q in question_data]
    context_data = [padded_vector(c, max_context_length) for c in context_data]
    question_data = [padded_vector(q, max_question_length) for q in question_data]
    #print('question uuids', question_uuid_data)
    #print('context data', context_data)




    #print('dirname, filename: ', dev_dirname, dev_filename)
    dataset = (question_data, context_data, question_masks, context_masks, question_uuid_data)
```

```
    # ========= Model-specific =========
    # You must change the following code to adjust to your model

    question_encoder = Encoder(size=FLAGS.state_size)
    context_encoder = Encoder(size=FLAGS.state_size)
    decoder = Decoder(output_size=FLAGS.output_size)

    qa = QASystem(question_encoder, context_encoder, decoder, embed_path, question_masks, context_masks)

    with tf.Session() as sess:
        train_dir = get_normalized_train_dir('train')
        #train_dir = './train/20170321_110723'
        initialize_model(sess, qa, train_dir)
        answers = generate_answers(sess, qa, dataset, rev_vocab)

        # write to json file to root dir
        with io.open('dev-prediction.json', 'w', encoding='utf-8') as f:
            f.write(unicode(json.dumps(answers, ensure_ascii=False)))


if __name__ == "__main__":
  tf.app.run()
```

Reading Comprehension on SQuAD

Stanford University (CS224N)
Meghana Rao, Brexton Pham, Zach Taylor
March 2017

Background

What are we looking at? *Question Answering*
Previous approaches

- Multiperspective Context Matching (MPCM) Model
- Dynamic Coattention Network (DCN)
- Match-LSTM with Answer Pointer

Datasets

For our dataset, we used the Stanford Question Answering Dataset (SQuAD), which contains context paragraphs that were extracted from a set of articles from Wikipedia. Humans then generated questions using that paragraph as a context, and selected a span from the same paragraph as the target answer.SQuAD is comprised of 100K question-answer pairs, along with a context paragraph. The answers are of variable lengths and require various forms of multi-sentence reasoning. The span of answers are accessible in a reference document.

Scenarios to Consider

1. Overfitting and underfitting
2. Accounting for unknown words in vocab
3. Amount of parameters and hyperparameter tuning

Problem Statement

In the SQuAD task, the goal is to predict an answer span tuple {as, ae} given a question of length n, q = {q1, q2, ..., qn}, and a supporting context paragraph p = {p1, p2, ..., pm} of length m. Thus, the model learns a function that, given a pair of sequences (q, p) returns a sequence of two scalar indices {as, ae} indicating the start position and end position of the answer in paragraph p, respectively.

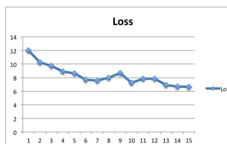
Methods/Algorithms/Models

1: A simple baseline in which we fed both the question and the context through an LSTM, and then for each hidden state in our context LSTM, we ran a simple dot product mechanism against the final question state to compute the probability that any index was the start or end index.

2: A model utilizing bidrectional LSTM's along with a question-aware context encoding. This model also had a simple attention mechanism, and we decoded the representations linearly.

3: A dynamic coattention model with an LSTM classifier as the decoder

Experimental Evaluation and Findings



FINAL TEST SET RESULTS (on codalab):

Model	F1 Score	EM Score
Separate Encodings	24.25	14.01

Predicted Answer	Correct Answer
hunting	no natural predators
claims	religious bigotry
March	Bonus March
1639	1639
Catholic	true apostolic succession
Toronto	Toronto

Visualization

Question: Where is the Center for New Religions located ?
Context Paragraph: Robert S. Wood has argued that the United States is a model for the world in terms of how a separation of church and state—no state-run or state-established church—is good for both the church and the state , allowing a variety of religions to flourish . Speaking at the Toronto-based Center for New Religions , Wood said that the freedom of conscience and assembly allowed under such a system has led to a " remarkable religiosity " in the United States that is n't present in other industrialized nations . Wood believes that the U.S. operates on " a sort of civic religion , " which includes a generally-shared belief in a creator who " expects better of us . " Beyond that , individuals are free to decide how they want to believe and fill in their own creeds and express their conscience . He calls this approach the " genius of religious sentiment in the United States .
"**Predicted Answer, Correct Answer:** Toronto, Toronto

Conclusion

After completing this project, we have gained perspective into the challenges involved with implementing a successful neural network for the SQuAD reading comprehension task. We ultimately demonstrated a functioning baseline model after investing into implementing more complex models that had long training durations and overfit the training data. Initial successes on the training data with more complex models did not perform well on the validation set and thus catalyzed a series of simplifications and debugging procedures to scale back the model to a slender and iterable scale. In retrospect, we wish to have implemented the barebones model first before attempting to implement more ambitious models. Nevertheless, we at least managed to get some positive performance eventually, and we learned an incredible amount about implementing neural network models for NLP. Also, we thought that even 13% of correct answers was a worthy accomplishment on such a difficult machine comprehension task.

Future Directions

For possible future directions, an obvious area of improvement would be changing the model to be able to predicted longer-than-one-word answers. This would involve careful experimentation and modification of the one model that we managed to get some meaningful performance out of. We could then graduate to implementing more advanced attention mechanisms. Potentially, we could learn what about our more advanced models didn't manage to translate to the evaluation set, and fix that element. In terms of other advanced models that we are interested in , answer pointers seem like a potentially valuable next step to take if we got the other elements working.