CSE P 517: Homework 4
Summarization, Recurrent Neural Networks, and Attention

Due Sunday Feb 28 2021 11:59pm

Hi class! In this assignment, we will be exploring an exciting modern task: summarization. We will be walking you through the steps a researcher takes to solving problems like this, from setting up evaluations and baseline models to training your own high-quality summarizer!

Please watch for updates on this assignment. It is completely new for this quarter, and we will update it with tips and fixes as we find bugs or places where students are confused.

Submission instructions As usual, submit 2 files on Canvas — **Code** (`HW4.zip`) and **Report** (`HW4.pdf`). The usual submission guidelines apply, which we omit now for brevity.

1 Setting up: Evaluation and Baselines

Summarization is a hot research topic in NLP today. There's plenty of data to train models with and it's quite *natural*–i.e. humans do it often and understand it well. Throughout this assignment, we will be walking you through the process of doing summarization research.

As an example, given the following document:

> The Irish sprinter clocked 21.86 seconds on Monday night with Poland's Mateusz Michalski second in 22.19. Smyth was back in action at the London Stadium a day after securing a fourth world title in the T13 100m. Meanwhile, Ballyclare's James Hamilton finished sixth in the T20 1500m final in a season's best of 4:01.78. The Great Britain runner was over eight seconds behind winner Michael Brannigan of the United States. Smyth will race for 200m gold in Tuesday night's final. The visually-impaired athlete from County Londonderry was a double Paralympic gold medallist at the same stadium in 2012. Smyth is unbeaten at Paralympic level since beginning his international career at the 2005 European Championships. Meanwhile, Ballyclare's James Hamilton will compete in the T20 1500m final later on Monday night.

We are trying to write a short summary that captures the most relevant information:

> Jason Smyth enjoyed a comfortable passage to the T13 200m final at the World Para-athletics Championships by cruising to a heat win in London.

Before we train any models though, we need to answer a basic question: *how will we know when our model is doing well?* Answering this requires 2 things: a **metric** to evaluate models–i.e. some way to score if they are doing the task well– and **baselines** to compare to, which are other models that we would like to beat.

Code We provide you with the base code required for this assignment. In the attached .zip file. For each questions, we specify which code you should be changing and writing. For Q1, you will be working with the script evaluate.py. You will work with code in the Q2 and Q3 directories for each for these questions. Generally, you should include any code you are asked to add or change in your submission, along with analysis and scores.

Unfortunately, this assignment is written in **python** and so you will need to write all code in python. Luckily, only small changes need to be made to each python file. If you are new to python, please attend office hours early to get lots of help!

Finally, **Q2 will take time, as you need to train multiple models.** Please start this question early enough that to train all models before the deadline (which may take up to 2 or 3 hours).

Generating an Azure Virtual Machine (VM) In this assignment, we highly suggest you use a Microsoft Azure virtual machine to run your code. This will allow TAs to help you more effectively. However, you are free to use your own computer if you would not like to set up Azure. Some setup may be slightly different in that case, but all models should still work.

First: accept the invitation to Azure Education sent out by TAs at the beginning of the semester. If you did not receive this email, please get in contact with TAs. To ensure you are correctly added to the Azure Education account for the course, click subscriptions (key icon) on the Azure home page. There should be a subscription for CSE517.

Second: You will be creating a virtual machine. Go to the Microsoft Azure homepage, click "Virtual Machines" and then on the Virtual Machines page click "Add" (virtual machine, not preset).

You will leave most fields default, other than the following:

- Give the Virtual machine a name (e.g. cse517)
- The Image should be Ubuntu Server 18.04 LTS - Gen 1
- The size should be Standard_D4s_v3 or Standard_D2s. The exact Size does not matter too much, as long as it's cheap enough that you can keep if for the full homework (2 weeks, given a budget of $100)

Other than this, you should be able to leave other fields default.

Once your virtual machine is created, follow the documentation on Microsoft Azure to connect by ssh, bastion, or RDP. The best option for you will likely depend on what computer you're connecting from.

Setting up Code and Environments We highly suggest you use the `conda` package/environment manager to run code in this assignment. You will need different packages and versions for each question, and Anaconda will make this very simple. There is plenty of documentation online (`https://conda.io/miniconda.html`) but we also provide specific instructions that should work on your Azure VM from above:

```
$ sudo apt-get update
$ sudo apt install unzip # You'll need this for opening the hw
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh # download
conda
$ bash Miniconda3-latest-Linux-x86_64.sh # After this command, scroll through licenses
and answer yes to all questions
$ source .profile
$ rm Miniconda3-latest-Linux-x86_64.sh
```

Now you should have `conda` setup! If you run into errors, we suggest you begin by Googling the error, particularly if you are using your own machine. `conda` is widely used and so most errors already have a solution, which TAs are unlikely to know offhand without also googling!

Next, we will setup the main environment for this homework using `conda`. Note, this environment will be used for Q1 and Q3, but we will need different packages for Q2 which will have its own environment.

To set up the main environment, you can run commands like:

```
$ conda create -n hw4 python=3.6
$ source activate hw4 # activate this environment.  You will need to do this before
running Q1 or Q3 code
$ # If using your own machine, you will need to use the appropriate pytorch installation
(see pytorch site)
$ conda install pytorch torchvision torchaudio cpuonly -c pytorch
$ pip install transformers
$ pip install datasets
$ pip install nltk
```

Once Anaconda is loaded, the environment is set up, and the assignment code is moved onto your Azure VM (or local machine if you are using this), you are ready to begin Q1.

1.1 Implementing Metrics

The first step in training summarization models is deciding on a way to evaluate them. The most common way evaluate models in Natural Language Genaration (NLG) is to compare what a model writes to what a human wrote. Does the machine-written summary look like the human-written summary?

Of course we if the machine writes exaclty the same summary as the human, we know it's good! But this is extremely unlikely. Even if we ask two humans to write a summary for the same document, they are very unlikely to be identical. So how do we compare how similar two different summaries are? We solve this by using **metrics** which can compare how much *overlap* there is between the machine-written summary and the human one. Metrics like this will often consider many different kinds of overlap: how many unigrams (words) are the same? How many bigrams? How many words are synonyms between the two?

Two very popular metrics are BLEU [4] and METEOR [1], which we will be using. Actually implementing them is quite complicated, but most NLP researchers never have to do this. Rather, we will incorporate them into an evaluation script, using existing implementations. You will start off by doing this! See evaluation.py for where you need to incorporate these metrics. It will be marked by Q1.1.

You will likely need to do some digging into documentation for the functions we provide, and do some processing on text before applying the metrics.

Once you are done, you can run:

```
python evaluate.py
```

We have included the output for one model from Q2 (cosine) in the generations folder, and so running this should output a METEOR score of 0.0667 and BLEU of 0.018 . Before implementing Q1.2 lead-5 and lead-10 should not return valid scores.

1.2 Very Basic Baseline: Lead-N

One of the most basic baselines researchers often try for summarization the lead-N baseline. This takes the first N words from the document you are trying to summarize, just using this as the summary. While it seems naive, many summarization datasets are based on news stories, which often DO include a decent summary as the first sentence.

You will implement this baseline in evaluate.py (Q1.2). Note that the documents **aren't processed**– they are in natural English. This means just calling `.split()` on a document will not actually break it up into words. Instead, you will need to call a specific function to break each document into words, take the first N, and then turn it back into a string (with a space between each word). See the evaluate.py for more information. Once this is implemented, run:

```
python evaluate.py
```

Which should now return valid scores for lead-5 and lead-10. Note, for lead-5 we get METEOR 0.0296 and BLEU 0.00075. Your numbers may be slightly different depending on package versions, but should be similar to this.

1.3 Deliverables

- (1 pt) Incorporate BLEU score into the evaluation script
- (1 pt) Incorporate METEOR score into the evaluation script
- (1 pt) Write the code implementing the lead-N baseline
- (1 pt) report metric values for lead-5, lead-10, and cosine

2 Baseline: RNN + Attention

In class, we talked about recurrent neural networks (RNNs, e.g. the LSTM [3]) which operate by reading text *sequentially*. We represent text with a *state vector* $\mathbf{h}$ which we update at every word (w_i):

$$\mathbf{h_i} = f(\mathbf{h_{i-1}}, w_i)$$

then use h_i to help us generate the next word, w_{i+1} when writing our summary! A common problem that comes up is, what if we need information from MANY words back to help us predict the next word? That information has to travel through our learned network f many times. Imagine that our model needs to know the first word the sequence (w_0) to predict w_{i+1}. When calculating $\mathbf{h_i}$ to make this prediction:

$$\mathbf{h_i} = f(f(...f(\mathbf{h_0}, w_0)..., w_{i-1}), w_i)$$

Any information about w_0 must travel through our neural network f, i times. Any errors in how f is calculated are amplified by applying every time, and we will likely not have any meaningful information about w_0, as it is corrupted over i steps. To combat this, we use the *attention* mechanism, which allows our current state $\mathbf{h_i}$ to update based on past states directly, using a weighted sum $\bar{\mathbf{h}}_\mathbf{i}$ that can directly access past states $\mathbf{h_j}$

$$\bar{\mathbf{h}}_\mathbf{i} = \sum_{j<i} \alpha(\mathbf{h_i}, \mathbf{h_j}) * \mathbf{h_j}$$

For instance, if we want information about the first state $\mathbf{h_0}$ this can come direction from this sum, rather than through many applications of f.

We can think of α paying *attention* to different parts of text from the past. The higher $\alpha(h_i, h_j)$, the more information we are taking from location j. We can use attention to make our representation h_i better, or even use it to copy word from the past in the Pointer Generator Network [6]. We will be using this architecture in question 2, and your job will be to implement different kinds of attention. **You do not need to have a complete understanding of RNNs or attention**, you will mainly be implementing scoring functions used for attention (below).

One important aspect of attention is that is should also be a valid probability distribution, that is:

$$\sum_{j<i} \alpha(h_i, h_j) = 1$$

An easy way to do this is by taking a scoring function $s(h_i, h_j)$ that we like, and pass it through a *softmax* function, which can turn any set of values into a probability distribution:

$$\alpha(h_i, h_j) = \frac{exp(s(h_i, h_j))}{\sum_{j'<i} exp(s(h_i, h_{j'}))}$$

This is a very useful method in machine learning, although you do not need to worry about it in this assignment. In question 2, you will be implementing different options for the scoring function used in attention.

Setup In this section, you will be working with code in the Q2 directory. Because different models often have different requirements, we will set up a virtual environment specifically for Q2. This will depend on what machine you're using, but with an azure virtual linux machine, this will require a series of commands like:

```
$ conda deactivate # deactivate any environment you might be in
$ conda create -n hw4q2 python=3.6
$ source activate hw4q2
$ cd ~/hw4/Q2 # change directories to q2 where code and requirements are
$ pip install -r requirements.txt
```

As long as you have no errors, you should be ready to start training RNNs! You can test the cosine scoring function by running:

```
$ bash Q2_0.sh
```

This should train for 10 epochs, then generate summaries for the test set (saved to hw4/generations/RNN_cosine.txt). If you run the evaluation script from Q1, this should now include `RNN_cosine`! (this should be roughly the same as `cosine` which we include with the code for testing–these are the same model but you are retraining the model and there is a bit of randomness).

For the rest of the assignment, you will be implementing different scoring functions in hw4/Q2/model.py. You will be working in the `get_attentions` function, and all locations to write code in are clearly marked.

HINTS:

- You will need to train multiple models for 30-40 minutes each on Azure VMs. The tmux command will be very useful here in case you lose your connection

2.1 Dot Product attention

Running Q2_0.sh runs attention with the cosine scoring function:

$$s_{cosine}(\mathbf{h_i}, \mathbf{h_j}) = \frac{\mathbf{h_i} \cdot \mathbf{h_j}}{||\mathbf{h_i}|| * ||\mathbf{h_j}||}$$

Where we are taking vector norms in the denominator. We are effectively taking a dot-product in the numerator then taking the magnitude of both vectors into account in the denominator.

A simpler version of this is just taking the dot product as the score:

$$s_{dot}(\mathbf{h_i}, \mathbf{h_j}) = \mathbf{h_i} \cdot \mathbf{h_j}$$

Implement s_{dot} in the location indicated at the top of Q2/model.py . You can use the existing implementation of s_{cosine} as a starting point. Once this is done, run the following command to train a model with this score and generate summaries!

```
$ bash Q2_1.sh
```

2.2 Cosine + Bias

While the cosine score is often preferred to the dot product, it has drawbacks. For instance, it is limited to the range [-1,1]. One simple way to get past this is by including a bias:

$$s_{cosine+bias}(\mathbf{h_i}, \mathbf{h_j}) = \frac{\mathbf{h_i} \cdot \mathbf{h_j}}{||\mathbf{h_i}|| * ||\mathbf{h_j}||} + bias$$

Where the bias is a constant shift to the range. Implement $s_{cosine+bias}$ in the marked spot in model.py, then run:

```
$ bash Q2_2.sh
```

to train a model with this attention and generate summaries.

2.3 Cosine + weight

One drawback of bias is that it is constant regardless of the input. We can fix this by calculating a bias specific to $\mathbf{h_i}$ using a weight vector $\mathbf{w}$:

$$s_{cosine+weight}(\mathbf{h_i}, \mathbf{h_j}) = \frac{\mathbf{h_i} \cdot \mathbf{h_j}}{||\mathbf{h_i}|| * ||\mathbf{h_j}||} + \mathbf{h_i} \cdot \mathbf{w}$$

Where $\mathbf{w}$ is a vector of weights used to calculate dynamic bias. Implement $s_{cosine+weight}$ in the marked spot in model.py, then run:

```
$ bash Q2_3.sh
```

to train a model with this attention and generate summaries.

2.4 Additive

Many different kinds of scoring functions have been considered for attention, not just those based on the dot product or cosine! For instance, we can use the addition of $\mathbf{h_i}$ and $\mathbf{h_i}$ along with a weight vector to calculate a completely different form of attention:

$$s_{additive}(\mathbf{h_i}, \mathbf{h_j}) = (\mathbf{h_i} + \mathbf{h_j}) \cdot \mathbf{w}$$

Where $\mathbf{w}$ is a vector of weights used to calculate attention. Implement $s_{additive}$ in the marked spot in model.py, then run:

```
$ bash Q2_4.sh
```

to train a model with this attention and generate summaries.

HINT: From Q2.1-Q2.4 we use matrix multiplication to do a dot product in the code. However, taking a dot product this way required $\mathbf{h_i}$ to be a "row" vector (shape = `[..., 1, n]`), and $\mathbf{h_j}$ to be a "column" vector (shape = `[...,n, 1]`). This will not work for adding them together, they will need to have the same shape. In this problem, you will need to take the *transpose* of one of them by calling `h.permute(0,1,3,2)`

In practice, we often let attention be calculated by complicated functions. We will explore a model based on this in Q3! (don't worry, you won't have to implement THAT attention)

2.5 Deliverables

- (4 pts) Implement each kind of attention above (1 pt each). Include this code in your submission.
- (1 pt) Run the evaluation script from Q1 and report your BLEU and METEOR scores for each kind of attention. Comment on which is the best, and whether you agree (looking at the summaries written in hw4/generations)

3 Bigger and Better Models

Recently, RNNs and other "small" models have seen very little attention (pun), despite being easy to train on your own computer. You may be familiar with the reason why: huge models like GPT-2 [5] and GPT-3 [2] do a much better job at language generation. You can see some amazing examples here `https://openai.com/blog/openai-api/`. Another connection to question 2: these very large models are

based on the *transformer architecture*, a kind of neural network that is based completely on attention. While attention was a useful way to augment the RNN in Q2, it is the entire basis for models like GPT-2 and GPT-3!

One issue however, is that these models need to first be trained on a huge amount of "general" data (e.g. text from the web) and then *finetuned* to do a specific task like summarization. Traditionally, finetuning means training all parameters of the model, like you did in question 2 but with a MUCH bigger model, making it intractable for anyone without a high-powered computer (i.e. GPUs).

Many researchers have been working against this though! A lot of recent work has been focussed on letting *anyone* tune a model like GPT-2, even on their laptop. We'll be using one of these techniques here: Prefix Tuning (cite). The intuition is, you don't have to train ALL of GPT-2 to do something cool. We can just train a very **small** part of the model and have a very **big** effect on performance.

Setup We can switch back to the `conda` environment used for Q1 to run Q3:

```
$ conda deactivate # deactivate any environment you might be in
$ conda activate hw4
```

3.1 Fixing the training loop

You will be changing code in Q3/summarize.py to fix the training loop. Currently, it only trains on the first example. However, it machine learning and NLP, we want to train on every example in the dataset. We also may want to do this multiple times (we call each time an epoch). Therefore, you should change the code to train over the full dataset, *n_epoch* times. The location where code should be changed is marked Q3.1.

Once the training loop is fixed, you can train a summarization model and generate summaries by running:

```
$ python summarize.py
```

We found this took about 10 minutes on the Azure VM, but it may depend on the specific machine. The script will run for 125 iterations, then generate outputs (each time you run it will generate outputs labelled `gpt2_<i>.txt` in the generation folder with increasing i. As you run experiments in 3.2 (And possibly 3.3) keep track of which generation corresponds to which time you run this script!

3.2 What effect does training have?

GPT-2 is a very strong model. It's possible that the training we are doing isn't having any effect, and it is already excellent at summarization. To test this, alter the training loop or parameters so that the model is not trained at all before generating summaries. How does the untrained model compare to the trained one?

Also, would more trianing help? See what happens when you train on the full training set twice.

You can also look at the *losses* array saved during training to see if anything is changing as training proceeds.

3.3 Overall comparison

Now that you have a wide range of models, do an overall comparison. How do scores compare between the different models? Do BLEU and METEOR tell us different things, or do they seem to agree? Look at the summaries in the generations/ folder. Do any of them look good? Can you find instances where one model scores worse, even though it seems to be writing better summaries?

We are not expecting a full lab report here. Rather, see where the results seem interesting to you (possibly including examples of summaries each system wrote) and talk about this a little bit.

Include scores for all your models.

3.4 Deliverables

- (1 pt) Code fixing the training loop
- (1 pt) Description of what you did to the code for Q3.2
- (1 pt) Scores for all models tested
- (2 pt) Your overall comparison from Q3.3

References

[1] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), 1997.

[4] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[5] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners.

[6] A. See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *ACL*, 2017.