

Assignment #4 - classic SMT and feed-forward neural networks.

[Submit Assignment](#)

Due No due date **Points** 30 **Submitting** a website url

Introduction

This assignment takes the "training wheels" off completely. You have the discretion to design the processing pipeline the way you like, as long as it conforms to the stated requirements and is well-documented enough for us to run and grade and to understand your design choices. You will also have to do some of your own research into the documentation of relevant Python modules and techniques in order to complete this assignment.

However, this is also a group assignment, meaning that you will have to segment the problem in such a way that you can effectively collaborate, and do so in a way that gives every member of your group a chance to contribute.

This assignment has **30 points** and **17 bonus points**. The official due date is **Monday, 2019 March 25, at 23:59**.

Classic SMT


As you saw in [Lecture 6](#) , the fundamental equation of "noisy-channel" statistical MT is:

$$\hat{t} = \operatorname{argmax}_t p(s|t)p(t)$$

using t for target language expression and s for source language expression rather than English and French as in the notes. For this assignment, you will model $p(t)$ as a trigram language model in a given target language and $p(s|t)$ as a "naïve" single-word-to-single-parallel-word translation model with the target language word predicting the the corresponding source language word.

However, you will implement each model as a feed-forward neural network with a two hidden layers in PyTorch with a softmax (logistic regression) function at the top layer of the network and maximize over the multiplied probabilities (or added log-probabilities...).

Feed-forward neural networks

We learned about feed forward neural networks in [Lecture 5](#) . Each hidden layer of the neural network is represented by a weight matrix of dimensionality N-by-M, where N is the dimensionality of the input vector and M is the dimensionality of the output. Between layers, you normally have a non-linear function so that high-dimensionality, linearly inseparable data sets can be e.g. classified.

For this assignment, you will develop your own FFNN implementation in PyTorch using matrix multiplications. You will use networks with two hidden layers and a softmax function turning the last output

into a probability. That means that given an input vector \mathbf{x} with dimensionality L , the first layer will be represented by a weight matrix \mathbf{W} of dimensionality L -by- N . The matrix product $\mathbf{W}\mathbf{x} + \mathbf{b}$ (don't forget the intercept!). But the second layer has its own N -by- M weights, \mathbf{U} , and bias \mathbf{c} , for an output dimensionality of M . With an internal sigmoid non-linearity and softmax output layer, we get the full equation:

$$P(y|\mathbf{x}) = \text{softmax}(\mathbf{U} \cdot \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c})$$

and what needs to be learned is everything but \mathbf{x} .

Collaboration and final submission

This is a collaborative group assignment. Very few "real" software projects are completed by a single person, especially in industry. We are dividing you up into groups of three. Each member of the group is expected to have a roughly equal opportunity to write code, document, and experiment.

In order to ensure fairness, your submission will be via GitHub, and we will take a look at the commit logs to ensure that everyone has committed a roughly equal amount of material (we do not expect perfect equality, but it should not look like one member is shouldering most of the burden or one member too little). A key goal is that every member of the group spend a fair share of time in the "driver's seat", so to speak. You will also include a markdown document, CONTRIBUTORS.md, in your submission, briefly stating what each member of your group did.

No prepared GitHub repository is provided for this assignment. One group member will create a GitHub project, and you will structure the assignment on your own.

As of this writing, there were 11 students signed up to do this project. That means three groups of three, and one group of two. The group of two will be graded out of 20 rather than 30 to be fair.

Preparation

PyTorch

Please individually read the PyTorch tutorials (you can also run and play around with the code yourself) on the following page other than the Data Parallelism tutorial:

[Link](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html) [_ \(https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html\)](https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html)

The data

You will be implementing French-to-English machine translation using a parallel corpus that is an extract of the UN corpus. The corpus will be found on mltgpu at:

/scratch/UN-english.txt.gz

/scratch/UN-french.txt.gz

Each line/sentence in the French file corresponds exactly to each line in the English file. You can normally access gzipped files using basic Python operations.

Part 1: preprocessing (2 points)

In this part, you will take the parallel corpus and randomly select, according to given parameters such as size (as command-line arguments), disjoint subsets of the original corpus as training and testing data. You can do any other text preprocessing you want here, as long as you document it (lowercasing and word tokenization recommended, but not by any means required). Finally, each line must be truncated to the same number of words as the parallel line in the other language; that is, you truncate a French sentence to the word length of the corresponding English sentence if the English sentence has fewer words, and vice versa. (This is because we are doing a very naïve word-by-word translation.)

For the truncation, consider the sentences:

```
This is where insurance comes in as a key component
in ensuring the healthy development of small and medium-sized
enterprises - a fact which is of paramount
importance to a country's political stability.
```

and

```
L'assurance devient alors un moyen clé de garantir le développement
sain des PME, un élément essentiel pour la stabilité politique
d'un pays.
```

Assuming you do no further tokenization, this is a rare case where the English has 33 words but the French only 22. So you must cut the last 11 words off the English sentence, so that every French word corresponds to an English word of equal index in the sentence. This will create massively wrong alignments, but this assignment will not deal with modeling distortion and fertility (see the notes), so we expect the output translations to be terrible.

Part 2: vectorization (3 points)

For the target (English) trigram language model $p(t)$, your code needs to collect the vocabulary for both the training and test data and turn it into vectors. You won't use one-hot vectors for your input layers, but you will need one-hot vectors for your output. For the input, you will use the pretrained word2vec 300-dimensional vectors that you can load via Python gensim. They are located in the scratch drive, here:

/scratch/GoogleNews-vectors-negative300.bin.gz

You can access the vectors via gensim's [KeyedVector](https://www.pydoc.io/pypi/gensim-3.2.0/autoapi/models/keyedvectors/index.html) class with the `binary=True` option.

Aside from the input vectors not being one-hot, from an input and output perspective, $p(t)$ will look just like assignment 3, but you will have implemented the "guts" of the model this time. You will still have to keep track of start symbols, possibly as random vectors. If there are English vocabulary items missing from the word2vec vectors, skip that word and the corresponding French word. (There are techniques to handle "out-of-vocabulary" words, but we won't try to smooth the distribution here.)

$p(s|t)$ will look a little different. It will take word2vec vectors from English as input, and predict one-hot vectors in French. So you will need, in other words, word2vec English vectors for the inputs of both models, one-hot English vectors for predicting the last word of a trigram, and one-hot French vectors for predicting the current word translation.

Part 3: simple PyTorch FFNN from scratch (13 points)

Develop the FFNN model described above. You can, if you want, do this as a class in a separate Python file which you can load as a module with "import". You will use the same kind of FFNN to separately train $p(t) = p(t_3 | t_1 t_2)$ and $p(s|t)$ as described above. You may not use the torch.nn layer classes, which automatically implement it for you; the idea is that you will manage the weights yourselves and therefore learn how the matrices work.

Since the output layer of the model is processed through softmax, the loss function you will use is categorical cross-entropy. You can use the Adam optimizer, which should optimize over the bottom layer weights \mathbf{W} and the bias weights. Do note that the output layer's matrix \mathbf{U} needs to have an output dimensionality the size of the one-hot vector vocabulary for which you are trying to calculate the loss (categorical cross-entropy).

Part 4: training and testing (7 points)

You will write a script to train the language model and the translation model and write out a model file containing both models.

Testing the model as a translation system is going to be rather inefficient. If you've already translated two words, you get the maximum $p(t) = p(t_3 | t_1 t_2)$ as the softmax over t_3 for free. But you need the t_3 that maximizes both the language model *and* the translation model $p(s|t)$ as the product of both (or sum of their log-probs), which may not be precisely the word that maximizes the language model alone. But the target word is the *input* of the translation model, so you would have to run the translation model for every word in the vocabulary, simply to extract the probability of the (fixed) source word for that target word. This is rather painful.

You can use the following procedure to estimate the argmax using a little shortcut devised on the spot for this assignment, where you run the entire vocabulary only for the first word:

1. For the first word in the sentence: since the French word is a given, you need to cycle through the entire English vocabulary as input to the translation model so that you find the English word that predicts the maximum probability for that French word.
2. Use the start symbol and that English word to predict the top 50 (you can make this a parameter) next English words.
3. Use those English words to maximize the translation model probability for the next French word.
4. Keep iterating until the end of the sentence.

Consider the example from Part 1, translating French into English. In that case, you will look up the probability distribution for the third English word after "<start> <start>". Then for *all* those words, you run the translation model to find which English word maximizes the probability of the French word "L'assurance" *when* you multiply its probability with the corresponding output of the source (English) trigram model. Now you know you have "<start> w" as the beginning of the next trigram, where w is the English word that was the argmax. Then you proceed, taking the top 50 words of the language model to test the translation model, until you run out of words in the parallel sentence.

Your decoder script should measure the accuracy, precision, recall, and F1 score of the words in English output, that is, how well they match the English translations (they will probably be bad, but that doesn't

matter to us now). It should, of course, also print out the translations, corresponding line-by-line to the French source.

Part 5: reporting and submission (5 points)

Document how to run the scripts in Markdown in README.md. Explain your design decisions. Document your experiments in training and testing, including experiments with changing the layer size of the FFNN models and other parameters. As above describe team member contributions at a general/high level in CONTRIBUTORS.md.

We should be able to test your scripts on other languages presuming we have the training and testing parallel texts for it in the same languages (for example, other UN languages) and a gensim-compatible word2vec file.

Part Bonus A: GPU (5 points)

It was OK to do the previous parts of the assignment in CPU mode. You can collect bonus points for adding a command-line option during training that pushes everything over to the GPU once training starts. Document your observations as to the performance in the README.md.

Part Bonus B: A different model entirely (12 points)

Use a single FFNN model to produce the translations, rather than two using the "hack" described above. Describe and document your model, and compare its performance to the "non-bonus" model.