Language Modelling with Recurrent Neural Networks[*]

Contents

Goals. The goal of this assignment is that you become familiar with implementations of some of the most common neural network models: the feed forward neural network, and the recurrent neural network. We will apply these networks to natural language modelling. Here are some online tutorials and blog posts about recurrent neural networks

- 'Recurrent Neural Networks for Beginners' by Camron Godbout
- 'Introduction to RNNs' by Denny Britz

Note: It may take long for the models in this assignment to be trained, if it is unreasonably slow on your computers it's ok to run them for fewer iterations. But of course this means you will get worse results and they will be harder to interpret.

Requirements. This assignment uses Python 3 and the library `numpy`, an immensely useful and popular library for working with vectors and matrices in Python. For a quick introduction to `numpy`, look at docs.scipy.org/doc/numpy-dev/user/quickstart.html. You also need to install the Natural Language Toolkit: the Python library `nltk`). You can install both libraries easily with `pip` or `conda`.

1 Feed Forward Neural Networks

In this part of the assignment, you will train a feed forward neural network to generate bi-grams (or, in other words, to predict the next word). The training

[*] Written by Samira Abnar (2018). Updated by Bas Cornelissen (2019).
Last updated on May 11, 2019.

data is a set of word pairs, where each word pair is a bi-gram. We treat the first word of each word pair as input and the second word as its corresponding output. Thus the network will learn to generate the next word after each given word.

In order to be able to feed words to a neural network, each word should be represented with a numerical vector. The simplest way to do this is to use one hot encoding (also known as *localist* encoding). This means that we will have a $M \times M$ identity matrix, where M is the size of the vocabulary and the i$^{\text{th}}$ row of the matrix is the representation of the i$^{\text{th}}$ word in the vocabulary.

Open the `FeedForward.py`. Here you see the `FeedForwardNeuralNetwork` class. This is an implementation of a neural network with one hidden layer. Take a look at the methods of the class. Read the comments if it's not clear what each method is doing.

For a neural network with one hidden layer, there are two weight matrices, one to map the input layer to the hidden layer, W_{in}, and another one to map the hidden layer to the output layer, W_{out}.

When we train the network for the next word prediction task, input and output are both word representations and they have the same dimensionality and the size of the hidden layer can be anything, it should not be very small or very large depending on the training data (the task).

QUESTION 1

The `init_params` method is where the weight matrices (the parameters of the model) are initialized. What should be the dimensions of each of these matrices? What is the total number of parameters of this model? You see the code for initializing W_{in}, use the same approach to initialize W_{out}.

In a feed-forward neural network (also known as multilayer perceptron), forward propagation is passing the input signal through the network while multiplying it by the respective weights to compute an output.

QUESTION 2

The `forward_pass` function is the method that defines how the output should be computed. Draw a graph that shows how the output is computed based on the input and the parameters in this model. Here is how you can instantiate and initialize a `FeedForwardNeuralNetwork`, and apply it to the data.

```
neural_network = FeedForwardNeuralNetwork(input_dim=input_dim,
                                          hidden_dim=hidden_dim,
                                          output_dim=output_dim)
output_from_layer1, output_from_layer2 =
                           neural_network.forward_pass(bigram_input)
```

QUESTION 3

What is the loss of the model before training?

Hint: use the `caculate_loss` method.

Now, let's train the model. You already know some search algorithms to find the optimal parameters of a model: in the previous assignment, you have seen Grid Search and Hill Climbing as optimization algorithms. You can imagine that it would very inefficient to use grid search optimize the parameters of a neural network (because of the large number of parameters). A common method to train neural networks is **back propagation** (backward propagation of errors) along with an optimization method such as gradient descent. The idea behind gradient descent is to change the parameters in a direction that decreases the loss. But instead of examining neighbour points randomly like in Hill Climbing, the gradient of the loss function is used to determine the direction of the change.

Here is a list of some of the resources that you can take a look at if you want to understand how back propagation really works.

- http://colah.github.io/posts/2015-08-Backprop/
- http://outlace.com/Beginner-Tutorial-Backpropagation/
- http://neuralnetworksanddeeplearning.com/chap2.html

There are three variants of back-propagation:

- **batch training:** all the training items are used to calculate an accumulated gradient for each weight and bias, and then each weight value is adjusted.
- **online training:** the gradients are calculated for each individual training item, and then each weight value is adjusted using the estimated gradients.
- **mini-batch training:** a batch of training items is used to compute the estimated gradients, and then each weight value is adjusted using the estimated gradients.

The `back_propagate_update` method in the `FeedForwardNeuralNetwork` class contains the code for updating parameters of the network with backpropagation. This method is called in the $train$ method.

You can consider the choice of one of these variants of backpropagation a parameter of the training procedure. In order to distinguish the training parameters from the model parameters, we refer to the former as *hyperparameters*. Other hyperparameters are the learning rate (how much we adjust the weights in each iteration) and the number of *epochs* or training iterations.

QUESTION 4

Call the `train_on_bigrams` method to train the model to predict the next word. Try it for three different sizes for the hidden layer and three different learning rates. Draw the loss per iteration plot for each experiment.

2 Word Embeddings

In this assignment, we have represented words as one hot vectors. Normally, the size of the vocabulary is too large and this is not an efficient encoding method. Actually, it is desired to have an *embedding space* with lower dimensionality. So, we need to have a $M \times E$ matrix, where E is the dimension of the embedding space. In the feed-forward model that you have trained here, the one hot vectors are mapped to the hidden state vector, applying W_{in}. Thus, basically, we can use W_{in} as an embedding matrix. For more background about word representations, you can look at this blogpost by Chris Olah.

QUESTION 5

For each word in the training data, compute its word embedding as `W_in[np.argmax(word_one_vector)]`. Feed the vectors to the T-SNE plotting function to plot the embeddings in a 2-dimensional space. Do you see any kind of regularity in this plot? Do this experiment for networks with different sizes of hidden layer (eg. 128, 256, 512).

```
input_vector = np.asarray([
    one_hot[i] for i in np.arange(len(index_to_word))
])
input_embeddings = np.dot(input_vector,W_in)
plot_distribution_t_SNE(input_embeddings,
                        np.repeat([1], len(index_to_word)),
                        [w.encode('utf-8') for w in index_to_word])
```

3 Recurrent Neural Networks

Open `RNN.py`. Here you see the `RNN` class. This is an implementation of a recurrent neural network with one hidden layer. Take a look at the methods of the class. Read the comments if it's not clear what each method is doing. In recurrent neural networks, in addition to the input, the last hidden state of the network is used to compute the new hidden state.

QUESTION 6

Look into the `init_params` method, where the weight matrices, parameters, of the model are initialized. What is each weight matrix for? What are the dimensions of each of these matrices? What is the total number of parameters of this model?

QUESTION 7

The `forward_pass` function is the method that defines how the output should be computed. Draw a graph that shows how the output is computed based on the input and the parameters.

QUESTION 8

Initialize an `RNN` and apply it on the training data (before training, using randomly initialized parameters). What is the loss? Now initialize the input weight matrix with the learned embedding matrix from the feed forward network. What is the loss now? compare the results.

Since in recurrent neural networks, at each time step the output depends also on the hidden state at the previous step, the error also needs to be propagated to the previous time step. This is why it is called **back propagation through time** (BPTT).

BONUS 8

The `back_propagation_through_time` method is the implementation of the BPTT for a recurrent neural network with one hidden layer. Change the code so that it propagates the error up to `max_back_steps`. Train the RNN with the sentences datasets for different values for `max_back_steps`: 1, 3, 5 and 10. First, run the experiment for one iteration and plot the loss function per number of seen sentences. Then run the model for multiple iterations (until the loss function doesn't decrease significantly) and plot the performance of the model per iteration.

BONUS 8

Use both `RNN` and `FeedForwardNeuralNetwork` to compute sentence prediction loss and generate sentences. Which one would you expect to perform better? Why? Are the results consistent with your executions? If not, why do you think this happens? Include performance plots and example sentences in your answer.

Hint 1: To compute sentence loss for the feed-forward neural network use the `calculate_sentence_loss` method defined in the `FeedForwardNeuralNetwork` class.

Hint 2: To generate sentences for both models use the `generate_sentence` method defined in each of the classes.