Language Modelling with Recurrent Neural Networks

Contents

Goals. In this computer lab, you become familiar with implementations of some common neural network models: the feed forward neural network, and the recurrent neural network. We will apply these networks to natural language modelling.

Requirements. This assignment uses Python 3 and the library `numpy`, an extremely useful and popular library for working with vectors and matrices in Python. Here you can find a quick introduction to Numpy. You also need to install the Natural Language Toolkit: the Python library `nltk`. You can install both libraries easily with `pip` or `conda`.

Note on training. 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.

1 Feed Forward Neural Networks

Let's jump right in. In the first part of the assignment, you will train a feed forward neural network to predict the next word, given the current word. In other words, you train it to generate bigrams. The idea is that we feed a word to the network and want it to output the next word. To get the network to do this,

Last updated on May 13, 2019. Written by Samira Abnar (2018). Updated by Bas Cornelissen (2019).

we need to *train* it. The training data will be a collection bigrams taken from some corpus. During training, we feed the first word of a bigram (the *input*) to the network, and see if the output matches the second word of the bigram (the *target*). Next we update the parameters of the network in such a way that predicting the target becomes more likely.

1.1 Data

Before we turn to the network, we need some training data. The file `DataReader.py` contains a class `DataReader` that reads out a text file, extracts (preprocessed) bigrams which serve as the training data. In this lab we will use Reddit comments as training data, which can be found in `materials/reddit-comments-2015-08.txt`:

```
from DataReader import DataReader
dr = DataReader('reddit-comments-2015-08.txt')
inputs, targets = dr.get_training_bigrams()
```

If you run this code, you might get the error `"Resource punkt not found"` because the `nltk` library uses datasets that you have to download first. The code should run after following the instructions in the error message:

```
$ python
>>> import nltk
>>> nltk.download('punkt')
```

You can see that `inputs` and `targets` are large matrices filled with zeros and ones. Why is that, if they should be words and the next words? Because we can't feed actual words (text) to a neural network: we have to represent them as numerical vectors. The simplest representation is a so called *one-hot encoding*. If we have vocabulary of M words, the one-hot representation of the i-th word is a vector of length M with only zeros, except for a 1 at at position i.

The `DataReader` implements all this: it computes training bigrams in a one-hot representation. You can en/decode words using the `dr.index_to_word` and `dr.word_to_index` dictionaries. For example, you can find the the 4-th input word using `dr.index_to_word[inputs[4].argmax()]`.

1.2 Architecture

If you are not familiar with artificial neural networks, do look for some introductions. Very briefly, a network consists of nodes organized in layers, such as an input layer, a hidden layer and an output layer. Every node has an activation (a number), which is computed from the activations of the nodes feeding into it, and the weights of the corresponding connections. Mathematically, the activation of a node is the weighted sum of the incoming activations – nearly, since we further apply a non-linearity such as $tanh$. It turns out that we can conveniently compute all activations in a layer at once using matrix multiplication. To give you an idea: if $x = (x_1, ..., x_n)$ is a vector with the activations of the input nodes, and W is a $m \times n$ matrix containing the weights of all connections between

this and the next layer, then the activations $y = (y_1, ..., y_m)$ of the next layer are $y = \tanh(W.x)$, where '.' is the dot-product.

The neural network we focus one here has one hidden layer, and therefore two weight matrices: W_{in}, which maps the input layer to the hidden layer; and W_{out}, which maps the hidden layer to the output layer. The weights are the trainable parameters of the network. In the case of next word prediction, both the input and output layers represent words and must have the same size (namely, the vocabulary size). The hidden layer, however, can have any dimensionality. Which size of the hidden layer works best is depends on the task. It is a so called *hyper-parameter* – more about that later.

In `FeedForward.py` you find the `FeedForwardNN` class which implements a feed forward neural network with one hidden layer. Try to understand how the codes work by looking at the methods of the class and reading the comments if it's not clear what each method is doing.

QUESTION 1

The `init_params` method initializes the weight matrices W_{in} and W_{out}.

- What should be the dimensions of these matrices?
- What is the total number of parameters of this model?
- You already see the code that initializes W_{in}. Adjust it to also initialize W_{out}.

In a feed-forward neural network forward propagation is passing the input signal through the network while multiplying it by the respective weights to compute an output. The `forward_pass` method implements all these computations for you: it computes the output of the network for a given input and model parameters. Here is how you can instantiate `FeedForwardNN`, and apply it to the data.

```
neural_network = FeedForwardNN(input_dim, hidden_dim, output_dim)
hidden_state, output_state = neural_network.forward_pass(inputs)
```

QUESTION 2

What is the loss of the model before training? Compute this using the `calculate_loss` method.

1.3 Training

Now, let's train the model. There are all kinds of search algorithms to find the optimal parameters of a model: grid search and hill climbing are two examples of such optimization algorithms. But it would be very inefficient to use something like grid search to optimize the parameters of a neural network, because it can have such a large number of parameters. Instead, neural networks are commonly trained using *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 neighbouring points randomly like in hill climbing, the gradient of the loss function is used to determine the direction of the change. We won't explain back propagation in detail here. If you are interested, look at this blogposts by Chris Olah, or this chapter by Michael Nielsen.

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.

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); the number of *epochs* or training iterations and architectural choices like the size of the hidden layer.

The `back_propagate_update` method in the `FeedForwardNN` class contains the code for updating parameters of the network with backpropagation. This method is called in the `train` method, which takes the hyperparameters as optional arguments. It returns an array with the loss after every iteration. The following code demonstrates how you can train the network:

```
# Load the training data; inputs = words, outputs = next words
dr = DataReader()
inputs, outputs = dr.get_trainig_bigrams()

# Initialize the neural network
vocabulary_size = len(inputs[0]) # = length of the first word-vector
neural_network = FeedForwardNN(
    input_dim=vocabulary_size,
    hidden_dim=256,
    output_dim=vocabulary_size)

# Train the network
trace = neural_network.train(
    inputs, outputs,
    num_iterations=100,
    learning_rate=0.01,
    batch_size=50)
```

After training, you can access the trained model parameters as `neural_network.W_in` and `neural_network.W_out`. If you want to use them later, it is convenient to store them using `np.save("my_filename.npy", neural_network.W_in)`, and later load them using `neural_network.W_in = np.load("my_filename.npy")`.

QUESTION 3

Use the code above to write a function that trains a feed forward neural network on next word prediction. It should take the hyperparameters `hidden_dim`, `learning_rate` and `batch_size` as arguments. Train models with three different hidden layer sizes and three different learning rates. Store the traces and afterwards plot the loss per iteration for all experiments. Can you explain what you find?

2 Word Embeddings

So far, we have represented words as one hot vectors. Although computing those vectors is easy, the representation is far from ideal. It fails to capture any linguistic structure: words with similar meanings need not have similar vectors, for example. All one-hot vectors are equally far apart. But you might imagine that the better your word vectors capture linguistic properties (such as meaning or grammatical categories), the better you can solve linguistic tasks.

Now, it turns out that neural language models internally often *learn* better word representations, even if we don't explicitly train them to do so. Take next word prediction: we update a network only based on its predictions, not on how it internally represents the words. But apparently the network can solve the task better if it also learns to represent the words in a linguistically more meaninguful way.

What is this magical internal representation? It is the vector of activations of the the first layer after the (one-hot) input layer. In complex networks with many layers, this layer is called the *embedding layer*: it embeds the one-hot vectors in some fixed-dimensional *embedding space*. In a well-trained model, this embedding space has typically picked up meaningful linguistic structure. This really is quite surprising – see this blogpost by Chris Olah for some examples.

Let's treat the hidden layer of our simple feed-forward network as an embedding layer and see if it reflects linguistic structure: do similar words cluster together? To do so, we compute the hidden state for every one-hot vector, and use the hidden states as embeddings. These are high-dimensional vectors, and to visualize them we use t-SNE, which is wonderfully illustrated in this article.

QUESTION 4

For each word in the training data, compute its word embedding: the corresponding hidden state. Feed the embeddings 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? Repeat this experiment with different sizes of hidden the layer (eg. 128, 256, 512):

3 Recurrent Neural Networks

Feed forward neural networks deal with one input at a time: they have no way to deal with sequences of data, like text: sequences of words. We now turn to *recurrent neural networks* which were designed to deal with sequential data. They add a so called recurrent connection to the network, from hidden state to itself. As a result, at every timestep the hidden state depends not only on the input at that timestep, but also on the hidden state at the previous timestep. This history-dependency allows it to capture sequential structure. For some quick introductions to recurrent neural networks, you can look at this blog post by Camron Godbout or this blog post by Denny Britz.

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.

QUESTION 5

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 6

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 7

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 7

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 7

Use both `RNN` and `FeedForwardNN` 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 `FeedForwardNN` class.

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