1 Introduction

In this assignment, we will experiment with recurrent neural networks on a POS tagging task using TensorFlow, one of the mainstream deep learning libraries. This assignment does NOT involve heavy implementation. However, we do need patience for the experiments as training neural networks takes time.

2 Preparation

Let us start by creating the working directory and download the skeleton code for the assignment via:

```bash
cd /your/working/directory
cp /local/kurs/ml/Assignment4.zip ./
unzip Assignment4.zip
cd Assignment4
```

Activate the virtual python environment by:

```bash
source /local/kurs/ml/2018/virtualenv/bin/activate
```

To have some basic knowledge of the task, let us open the data file by:

```bash
less English/dev.txt
```

In this assignment, we predict POS tags using sequence of words as input. The data set is built from the UD English treebank (http://universaldependencies.org/). Now run the script by:

```bash
python tagger.py
```

This will take quite a few minutes. Finally, you will get an accuracy of around 0.87 on the test set. To be fair, this accuracy is already somewhat reasonable. Nonetheless, we will start from this very basic model and incrementally improve it in this assignment.

Now open `tagger.py` in your favourite python script editor (pyCharm, Vim, Emacs, etc), try to understand the structure and functionality of the script. We predict the POS tag by first mapping the words into embeddings and then feeding the word embeddings into a recurrent neural network. The output of the recurrent layer is passed into a time-wise fully-connected dense layer that functions as the output interface.

3 Fully-connected Dense Layer (5 p)

Let’s take a closer look at how the computational graph is defined in `tagger.py` and identify different components of the neural networks. What are the input and output of the `DenseLayer` layer? What are the shapes of the input and output tensors?

`DenseLayer` is a regular feed-forward neural network. Open `layers.py` and try to understand how it is defined. What is the purpose of adding non-linear activations to the linear product?

4 Long-short Term Memory (LSTM) Cell (5 p)

Open `layers.py` and check how the recurrent layer RNN is defined. Currently, the recurrent layer in our neural networks employs `BasicRNN` as the recurrent cell. We want to employ the LSTM cell instead. Check how `tf.nn.rnn_cell.LSTMCell` is defined on the TensorFlow website. In addition, understand how `BasicRNN` is defined and called in `RNN` and then add `tf.nn.rnn_cell.LSTMCell` similarly.

Go back to `tagger.py` and change the parameter `cell_type` of the RNN layer from ’basic’ to ’LSTM’. Run the tagger again. Is the accuracy significantly improved? Why or why not? How about the training time?
5 Bidirectional RNN (20 p)

Our current recurrent layer is one-directional. First, let’s take a look at how the recurrent wrapper `tf.nn.dynamic_rnn` in the `call()` function of `RNN` is defined. Similarly, we will apply `tf.nn.bidirectional_dynamic_rnn` to implement bi-directional RNN. Read carefully on how `tf.nn.bidirectional_dynamic_rnn` works.

You may notice that `tf.nn.bidirectional_dynamic_rnn` requires two recurrent cells. Thus, we need to define the corresponding backward RNN cells in the `init()` function of `RNN` first. Next, add the bidirectional wrapper in `call()`.

Unfortunately, we cannot feed the output of `bidirectional_dynamic_rnn` to the following layer as for the normal `dynamic_rnn`. What is the difference between the outputs of these two wrappers? What are their connections? Find the answers on the TensorFlow website.

We need to concatenate the outputs of the forward and backward recurrent layers time-wise before passing them to the inference layer using `tf.concat`, one of the most useful functions. Find out how `tf.concat` works, pay attention to the shape of the applied tensor and the `axis` parameter.

Now, go back to `tagger.py`, change the parameter `rnn_type` of the `RNN` layer from ‘normal’ to ‘BiRNN’. Additionally, we need to change `input_dim` of `DenseLayer` to `rnn.state_size * 2`. Why?

If everything is done properly, run `tagger.py` again. Does the bi-directional RNN work better? Why or why not?

6 Word Embeddings (20 p)

Our neural network uses pre-trained word embeddings as the core feature. Open the embedding file under the `embeddings/` folder to see how it looks. Now, pick your favourite word in the embedding file and compare its vector representations with the other words’. Find its most similar word in terms of cosine similarity. Are they intuitively similar to you? Why or why not?

For efficiently computing the cosine similarity, we can reuse the function `readEmbeddings()` in `reader.py` and store the vectors in numpy arrays. The cosine similarity between two vectors can be computed by:

```python
>>> from scipy import spatial
>>> similarity = 1 - spatial.distance.cosine(v1, v2)
```

Let us change the `embPath` parameter to `None` so that the model will use randomly initialised embeddings instead of the pre-trained ones. Run `tagger.py` and describe what you observe. Change `embPath` back to the embedding path when the run is finished.

7 Optimisers (10 p)

The standard gradient descent optimiser `tf.train.GradientDescentOptimizer()` is so far used for all the experiments with the constant learning rate of 0.02. Take a look at [https://www.tensorflow.org/api_guides/python/train#Optimizers](https://www.tensorflow.org/api_guides/python/train#Optimizers) for some alternative optimisers. Choose one additional optimiser from the list and substitute `tf.train.GradientDescentOptimizer()`. Run the script and test if it works differently in terms of accuracy and training time. For some optimisers, for instances Adam and Adadelta, use much smaller learning rates (comparable to the default values). You may notice that using some optimisers takes significantly longer time for training.

Let’s change our optimiser to `tf.train.AdagradOptimizer()` for the following experiments as optimisers with adaptive learning rates normally work better for recurrent neural networks.

8 Dropout (15 p)

Dropout is a commonly used regularisation technique. Check [https://www.tensorflow.org/api_docs/python/tf/nn/dropout](https://www.tensorflow.org/api_docs/python/tf/nn/dropout) and figure out how `tf.nn.dropout` works and apply it to the output of your recurrent layer. (Note: `keep_prob = 1 - dropout_rate`) Dropout is only applied at the training stage while it should be turned off when decoding. The best way to integrate dropout in the model is to define `tf.placeholder()` and add it to `batch.train()` and `batch.predict()` along with the dropout value if needed. Try different dropout rates and see how they affect the training of our model.
9  Hyper-Parameter Tuning (20 p)

There are many hyper-parameters that influence the performance of our neural network. Try to play around with them by:

- Changing RNN state size
- Adding more recurrent layers
- Changing batch size
- Changing learning rate
- Changing dropout rate
- Running for more epochs

First, change the parameters individually and then try to find the best combination. Comment out the last three lines of `tagger.py` while you tune the parameters as we reserve the test set for final evaluation. Try to obtain higher scores on the development set. What accuracy does your best model achieve on the development set? Try to interpret why the corresponding configuration works better than the others. Save your best model in `tagger.py`. Briefly introduce the parameter tuning procedure.

Apply your best model on the test set, what accuracy does it achieve?

10  Further Development (5 p)

The state-of-the-art accuracy on the test data is around 0.951 (do not try to beat it within this assignment). Review the lecture notes, reading materials, summarise your experimental results as well as read some relevant papers if you think necessary. Discuss what else can be done to obtain further improvements to the best of your knowledge.

Submission

You should submit a short written report (4-5 pages) together with your modified `tagger.py` and `layers.py`. Try to solve all the problems and answer the questions in your report. You need at least 70 points to pass this assignment and 90 points to pass with distinction (VG). Please upload all the required materials to `Studentportalen` before June 1, 2018.