**CRF-NN Interface Specifications**

CRF needs to contain two types of weight vectors:

* The “main” weights, which has 2 parts:
  + Neural: the weight is computed by NN – let’s call this **EXTERNAL** neuralweight. CRF knows how to compute derivative w.r.t to this weight using dynamic programming.
  + Non-neural: any other features not interacting with NN
* **INTERNAL** neural weights
  + The flattened 1D weight vector of the internal neural network
  + CRF does NOT know how to compute the gradient for these weights. It has to call the BACKWARD API for this, passing gradients for **EXTERNAL** **neural** weights for NN’s backpropagation.
  + CRF holds a reference to INTERNAL neural weight vector for one reason: **unified optimization***.* This means the **INTERNAL neural** weight vector is concatenated with **non-neural** weight vector and then is updated together by calling the optimizer (e.g., LBFGS).

Currently this assumes full-batch mode. The overall flow of CRF-NN is the following:

1. CRF creates and initializes the back-end neural network using INIT API, also sends to NN all information for all hyperedges and instances.
2. CRF randomly initializes non-neural weights + INTERNAL neural weights.
3. CRF calls FORWARD API, passing the most up-to-date INTERNAL neural weights.
4. NN overwrites its internal weights with the ones sent by CRF.
5. NN iterates through all hyperedges and instances, computing their scores then passing them to CRF.
6. CRF computes new objective value and gradients (non-neural and EXTERNAL neural).
7. CRF calls BACKWARD, passing the gradients for EXTERNAL neural to NN.
8. NN computes INTERNAL neural gradients based on EXTERNAL neural gradients, then pass INTERNAL gradients to CRF.
9. CRF calls optimizer to make a joint update on concatenated non-neural + INTERNAL neural weights.
10. Repeat from 3 until convergence.

The interface contains 3 main APIs. Below I will describe the input arguments (*Args*) and its functionality (*Actions*) for each API. Data communicationwill be done through **ZeroMQ sockets** in the form of **JSON** format (essentially a dictionary of key-value pairs).

1. **Network Initialization**

*Description*:

Creates a neural network back-end with given specifications. This API is to be called **once** at the start before training begins.

*Args*:

* **Hyperedge** definition:
  + An array containing the dimension for each field
  + An array containing the input type for each field:
    - 0 = one-hot
    - n = lookup table with embedding size n. Some of the lookup tables can be shared among inputs (e.g., word input).

This describes what kind of information we want to include from the hyperedge. For example, we may include (current word+previous word+any additional tag). Then we will create the following JSON entry:

hyperedge: {

vocabSize: [10000, 10000, 40]

embeddingSize: [100, 100, 0]

}

* **Instance** definition.
  + Structure: independent, linear, or tree
  + An array of instances containing:
    - The input sentence
    - (Optional) A structure of the instance’s observation
      * Only applicable for linear or tree structure
      * Tree structure can be obtained e.g. from a syntactic parser. The span label can be ignored since we are only interested in the structure of the observation.
    - An array of hyperedges for each instance and its position in the structure (if any).

This describes the structure of the instance. For example,

instanceStructure: “tree”,

instances: [

{// first instance

// assume the following parse tree:

// (S (NP (NNP John)) (VP (VBZ likes) (NP (NNP Mary))))

observation: “John likes Mary”,

structure: [[0,3], [0,1], [1,3], [1,2], [2,3]],

hyperedges: [[“John”,[0,1]],[”likes”,[1,3]],[“likes”,[1,2]], …]

},

{// second instance

observation: […],

structure: […],

hyperedges: […]

},

…

]

* **Network** definition. This describes the NN’s parameters.
  + - Number of output nodes (equals number of labels in CRF)
    - Number of layers
    - Number of hidden units per layer
    - Activation type (e.g, ReLU)
    - Temporality (static, dynamic)
      * Dynamic will take into account the instance’s structure, resulting in a recursive-structured neural network (either with RNN/LSTM/GRU memory). Static will ignore the structure information, resulting in a multi-layer perceptron.

*Actions*:

Creates a neural network based on Args. Returns the number of INTERNAL neural weights so that CRF can (randomly) initialize the INTERNAL neural weights.

1. **Forward**

*Args*:

* Updated INTERNAL weight vector **newWeights**

*Actions*:

* Overwrite NN’s internal weights with the ones provided in Args:

NN.weights = **newWeights**

* Return an array ***scores***, computed as follows:

for each instance *i*

for each hyperedge *h* in *i* // the iteration order may depend on i’s structure

***scores***[i,h] = NN.forward((*i*,*h*))

return ***scores***

1. **Backward**

*Args*:

* EXTERNAL gradients **grads**

*Actions*:

* Return INTERNAL gradients, computed as follows:

for each instance i

for each hyperedge *h* in *i*

NN.backward((*i*,*h*), ***grads***[i,h]) // backward() internally updates NN.grads

returnNN.grads