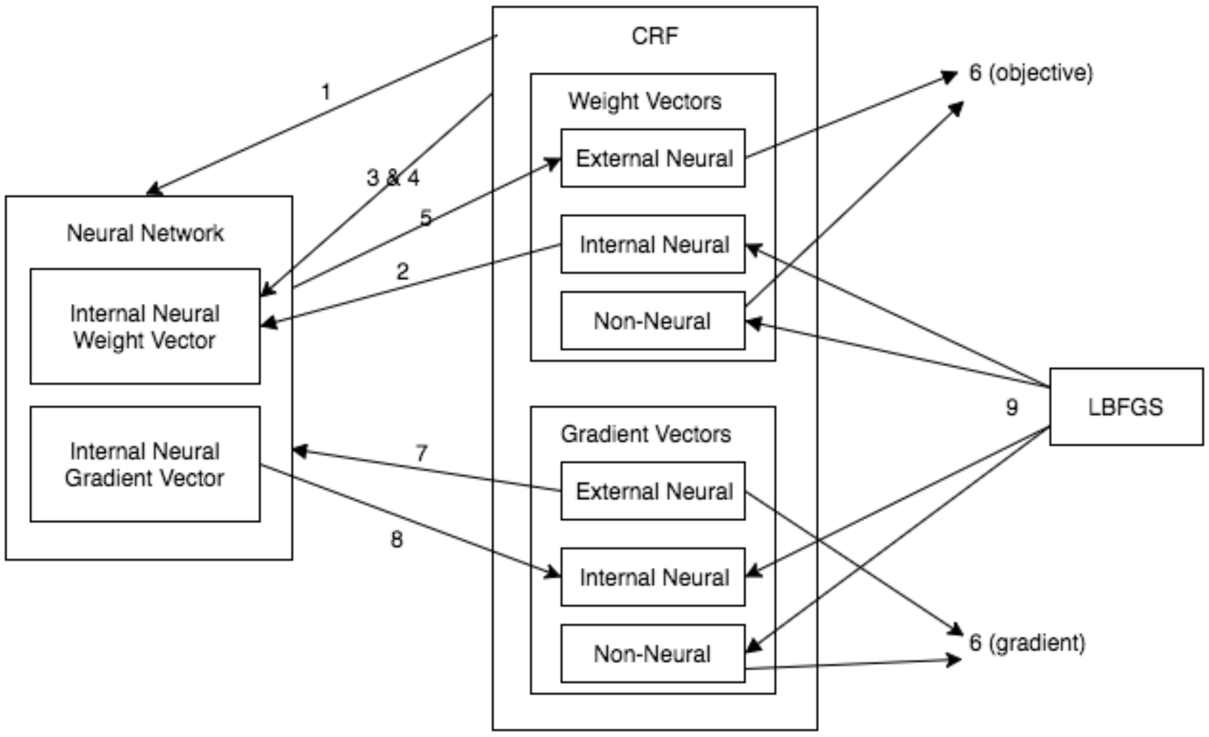
CRF-NN Interface Specifications



CRF needs to contain three types of weight and gradient vectors:

- External Neural
  - The weights computed by NN.
  - CRF knows how to compute derivative w.r.t to this weight using dynamic programming.
- Internal Neural
  - The flattened 1D weight vector of the internal neural network.
  - CRF holds a reference to Internal Neural for **unified optimization**. This means the Internal Neural and Non-Neural are updated together by the optimizer (e.g., LBFGS). These weights are not involved when computing CRF's objective function.
  - CRF does not know how to compute the gradient for these weights. It has to call BACKWARD and pass the gradients for External Neural for NN's backpropagation.
- Non-Neural: any other features not interacting with NN

The overall flow of CRF-NN is the following: (refer to the diagram above for illustration)

1. CRF creates and initializes the back-end neural network using INIT, and sends hyperedge information to NN.
2. CRF randomly initializes Internal Neural weight vectors.
3. CRF calls FORWARD and passes the most up-to-date Internal Neural weights.
4. NN overwrites its internal weights with the ones sent by CRF.
5. NN computes External Neural weights for each hyperedge.
6. CRF computes new objective value and gradients (Non-Neural and External Neural).
7. CRF calls BACKWARD and passes the gradients for External Neural.

8. NN computes Internal Neural gradients, then passes them to CRF.
9. CRF calls optimizer to update Non-Neural + Internal Neural weights.
10. Repeat from 3 until convergence.

The interface contains 3 main APIs. Below I will describe the input arguments (*Arguments*) and its functionality (*Actions*) for each API. Data communication will 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.

Arguments:

- Hyperedge definition.
    - An array containing the dimension for each field
    - An array containing the desired embedding size for each field:
        - 0 = one-hot
        - n = lookup table with embedding size n
    - An array containing unique hyperedges that needs to be computed by NN.

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

```
{
      vocabSize: [10000, 10000, 40]
      embeddingSize: [100, 100, 0]
      vocab: [[1,2,3], …]
}
```

    For example, the `vocab` above contains a tuple (1,2,3). This corresponds to a hyperedge identified by:
        current word = word_1
        previous word = word_2
        tag = tag_3

- Network definition.
    - Number of output nodes (e.g., number of labels in CRF)
    - Number of layers
    - Number of hidden units per layer
    - Activation type (e.g, ReLU)

Actions:
Creates a neural network back-end. Returns the dimension of Internal Neural weights vectors.

2. **Forward**

Arguments:
- Weights of Internal Neural (***weights***)

Actions:
- Overwrites NN's internal weights. *Pseudocode*:

```
NN.weights = weights
```

- Returns an array ***scores***. *Pseudocode*:

```
for each hyperedge h
    scores[h] = NN.forward(h)
return scores
```

3. **Backward**

Args:
- Gradients of External Neural (***grads***)

Actions:
- Returns gradients of Internal Neural (**NN.grads**). *Pseudocode*:

```
for each hyperedge h
    NN.backward(h, grads[h]) // backward() internally updates NN.grads
return NN.grads
```

Code

The main controller class is called **NNCRFInterface**. This is an abstract class which contains a couple abstract methods that need to be implemented. Refer to the source code for more information. Comments are provided for each methods and the "steps" mentioned there correspond to the diagram shown in the first page of this document.

NNCRFGlobalNetworkParam is one example of a concrete implementation (a subclass) of NNCRFInterface. This example is a part of the statistical framework in our research group. Internally, it interacts with GlobalNetworkParam, a model class in the framework which stores weights and gradients information. Hence, for other frameworks, we would need to implement something similar.

RemoteNN interacts with the neural network back-end in Torch. It takes care of preparing the request in JSON format and fetching weights/gradients from NNCRFInterface.