# NNBlocks: A Deep Learning Framework for Computational Linguistics Neural Network Models

[*]Frederico Tommasi Caroli, [*]João Carlos Pereira da Silva, [**]André Freitas, [**]Siegfried Handschuh
[*]Universidade Federal do Rio de Janeiro, [**]Universität Passau
[*]Rio de Janeiro, [**]Passau
ftcaroli@gmail.com, jcps@dcc.ufrj.br, andre.freitas@uni-passau.de, siegfried.handschuh@uni-passau.de

**Abstract**

Lately, with the success of Deep Learning techniques in some computational linguistics tasks, many researchers want to explore new models for their linguistics applications. These models tend to be very different from what standard Neural Networks look like, limiting the possibility to use standard Neural Networks frameworks. This work presents NNBlocks, a new framework written in Python to build and train Neural Networks that are not constrained by a specific kind of architecture, making it possible to use it in computational linguistics.

**Keywords:** Deep Learning, Artificial Neural Network, Computational Linguistics

## 1. Introduction

Neural Networks (NN) are a kind of statistical learning methods that has been gaining a lot of attention with the appearance of a variety of techniques that make possible the so-called Deep Learning (DL). DL happens when a statistical model, normally a NN, learns layers of high-level abstractions in the data. This idea became popular in 2006(Hinton et al., 2006), but just recently this kind of learning has being applied successfully in computational linguistics. This is partially driven by the fact that linguistics features are highly complex and tend to not fit in traditional NNs architectures. The most notable example is Multilayer Perceptron, that cannot handle variable sized inputs.

With the creation of new kinds of NNs architectures, researchers can fit their linguistics tasks and experiment with them, but this comes with a great load of implementing the model and training it from scratch, since standard NNs frameworks are not always ready to deal with architectures that are very different from the standard ones applied for other tasks.

NNBlocks is a Deep Learning framework written in Python that aims to solve this problem, being very expandable and easy to use, while providing state-of-the-art training techniques without a single modification to the model. The easy training of NNs is achieved by using Theano(Bergstra et al., 2010), a math expression compiler and automatic symbolic derivation framework, as workhorse of NNBlocks. This also gives NNBlocks the ability to run on GPU without any modifications to the code. NNBlocks makes it possible to implement and test NNs that are very different from standard ones.

The first section presents some related work, the second section discusses NNs architectures used for computational linguistics tasks, the third section presents the framework and the fourth concludes discussing how NNBlocks can achieve the previously discussed architectures.

## 2. Related Work

Some Deep Learning frameworks with variable degrees of flexibility exists. In this section we will review some of them and discuss what points could be improved.

The first framework we will talk about is Lasagne.[1] This framework, like others, has an approach similar to NNBlocks, in which processing blocks are connected with each other to create a complete NN. This framework is able to build a complex processing graph with its blocks and is somewhat easy to be extended with new blocks. The main flaw in this framework is the overexposure to Theano mechanisms. Some user wanting to use the framework is obligated to learn how Theano works too, which can be time costly. The framework also lacks ways to easily handle variable sized data and a Recursive NN implementation. Lasagne is probably the most similar work to NNBlocks.

Another framework normally used is Chainer.[2] This framework has different objectives than NNBlocks. It is not made to be easily extended. It provides fast implementations to known architectures and is easy to use. NNBlocks is different in the sense that is made to be flexible enough to be easily extended. A framework similar to Chainer is Torch7.[3] It is similar in the sense that is made to be easy to use and fast, but not flexible and extendable.

One final framework that is worth mentioning is Google's TensorFlow.[4] This framework is similar to Theano and has all of the same objectives. The problem with using TensorFlow directly is that much of the implementation work for NNs will still be necessary. The advantages of this framework is that is easier to use than Theano and is made to run on distributed environments. One thing that could be considered as future work for NNBlocks is the usage of TensorFlow as an alternative workhorse.

## 3. Linguistics Neural Network Models

Most times the first thing noticed when trying to use NNs in linguistics tasks is that the standard Multilayer Perceptron NN (Figure 1) cannot handle variable sized inputs. This limits greatly its ability to deal with things like sentences,

[1]`http://lasagne.readthedocs.org/en/latest/`
[2]`http://chainer.org/`
[3]`http://torch.ch/`
[4]`https://www.tensorflow.org/`

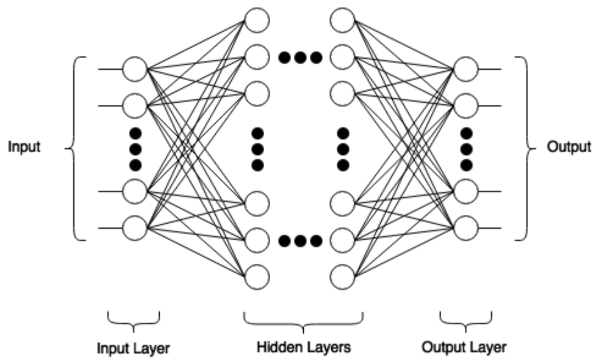

Figure 1: Multilayer Perceptron

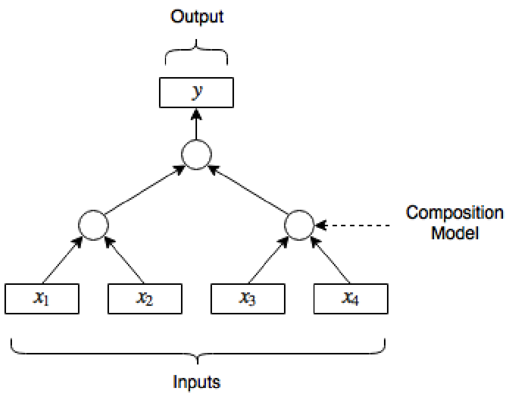

Figure 2: Recursive Neural Network

that are highly variable in size. Other models, such as the Recursive NN (Figure 2)(Goller and Kuchler, 1996) finds success dealing with parse trees of sentences naturally. Although the latter model found its place in linguistics tasks, it is rarely implemented in NNs frameworks. One framework that implements a variation of this architecture is DL4J.[5]

Here we present some popular architectures for NNs in computational linguistics, explaining briefly how they work and what kind of results they achieve.

## 3.1. Continuous Bag-of-Words

The Continuous Bag-of-Words (CBOW)(Mikolov et al., 2013) is a model for estimating continuous representation of words, often called *word vectors*. This neural network has as input $C$ one-hot vectors of size $|V|$, where $C$ is the context window size and $V$ is the vocabulary. The output of this NN is another vector of size $|V|$.

The objective of this model is to predict:

$$P(w_t|w_{t-C}, w_{t-C+1}, ..., w_{t-1}, w_{t+1}, ..., w_{t+C-1}, w_{t+C})$$

where $w_j$ are words, represented as one-hot vectors in the model. In other words, the model tries to determine the probability of the occurrence of a word given a context window. It does it by first applying a projection layer that averages the input vectors and feeds this average to $D$ neurons,

[5]http://deeplearning4j.org/recursiveneuraltensornetwork.html

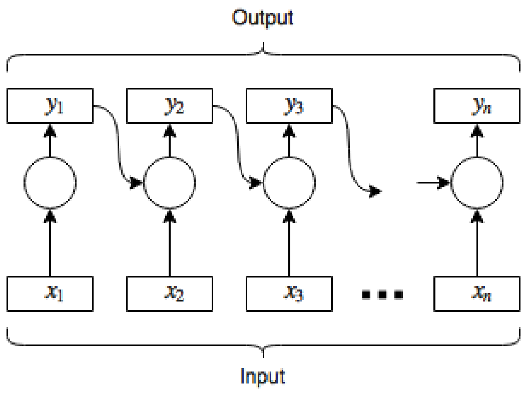

Figure 3: Recurrent Neural Network

$D$ being the size of the wanted word vectors:

$$h = \frac{1}{2C} W(\sum_{i=1}^{C} w_{t-i} + \sum_{i=1}^{C} w_{t+i})$$

where $W$ is the weight matrix for the $D$ neurons. Next the $h$ vector is fed to a Softmax classifier that gives the output probabilities.

After training the model with the objective of predicting a word given its context, the matrix $W$ has $|V|$ column vectors that capture contextual information about each word in the vocabulary. These vectors, the word vectors, can be used as features for other tasks that require semantic and syntactic information about words.

## 3.2. Recursive Neural Network

Recursive NN (Figure 2) uses a binary tree structure to map its variable size input to a single output. This tree structure is great for using parse trees of sentences and word vectors as leafs. The model recursively applies a composition function following the tree structure. This kind of NN has great results in compositionality tasks. The simplest Recursive NN is the one where a parent vector $p$ is

$$p = \sigma(W \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b) \qquad (1)$$

where $c_1$ and $c_2$ are child vectors, $W$ is a weight matrix, $b$ is a bias vector and $\sigma$ is an activation function, e.g. the sigmoid function. The result of this recursive process, the last parent vector, can then be used as a semantic feature of the sentence. These kind of compositional vectors have great results in parsing(Socher et al., 2013a), paraphrase detection(Socher et al., 2011), entities relationship classification(Socher et al., 2012), sentiment detection(Socher et al., 2013b) and more.

Normally, the simple composition function in (1) is not enough to fully capture semantic relationships between words or phrases. One Recursive NN that tries to overcome this is the Recursive Neural Tensor Network(Socher et al., 2013b). This Recursive NN uses layers of a tensor $V$ to compose each dimension of the parent vector.

## 3.3. Recurrent Neural Network

Recurrent NNs (Figure 3), just like the Recursive NN, can handle variable sized inputs. The idea behind this kind of

network is to keep some kind of memory of previous results as features for the next input. And just like the Recursive NN, this NN is a good way to capture semantic aspects of sentences. This kind of network has great results with language models(Mikolov, 2012), compositionality(Le and Zuidema, 2015) and more.
The simplest case for Recurrent NN is when the recurrence function is:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$$

where $W_h$ and $W_x$ are weight matrices, $h_{t-1}$ is the result of the previous recurrence and $x_t$ is the current input. Other more complicated recurrence functions exists, the Long Short Term Memory(Hochreiter and Schmidhuber, 1997) being a particularly popular choice.

## 4. NNBlocks

Two things that all of models presented above have in common are: (a) they have some very different aspects from common Multilayer Perceptrons and (b) they all have a lot of architectural variations depending on the task. This makes these kinds of models very hard to be served by NNs frameworks, because such frameworks cannot implement an architectural variation just for the sake of a single task.
NNBlocks deals with these problems by serving modular operations of the right complexity for computational linguistics and freedom to combine these operations in a lot of ways. NNBlocks does this while providing simple ways to extend the framework's operations in few lines of code. How NNBlocks achieves this is described in this section.

### 4.1. Models

NNBlocks builds NNs architectures using objects that extends the class Model. This class represents the NNs operations discussed previously. These basic blocks will have their outputs connected to another Model's inputs. Basically anything that has any number (possibly 0) of inputs and outputs in the network is a Model.
A Model can also have any number of optimizable parameters too. These are normally connections' weights, but it can be anything that plays a role in the network's computation.
NNBlocks comes with some out-of-the-box models, some of them:

- Input layer
- Perceptron layer
- Softmax layer
- Recurrent and Recursive NNs – These are highly customizable versions of the simple standard Recurrent and Recursive NNs, as we shall see.
- Convolutional NN
- LSTM recurrence

And if the already implemented Models are not enough for a wanted architecture, as in any realistic case, NNBlocks provides a nice way of creating a custom Model without

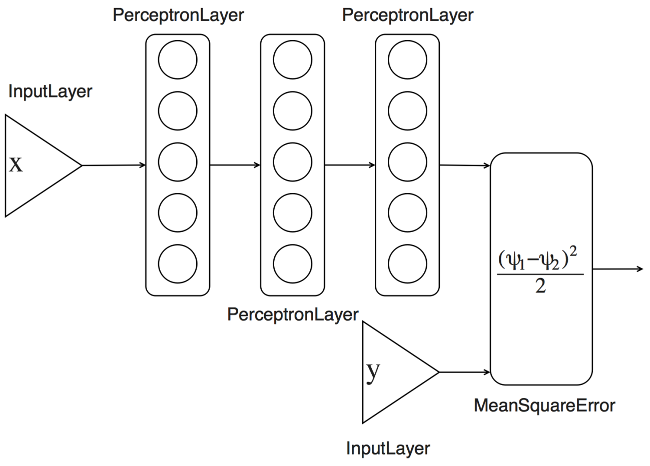

Figure 4: Connecting Models

the need of extending the Model class. The next example shows how to create and connect a Model that simply multiplies a number by 2.

```
import nnb

#Function to be turned into a model
times_2 = lambda x: 2 * x

#Blocks
X = nnb.InputLayer(ndim=0)
TIMES2 = nnb.CustomModel(fn=times_2)
MODEL = X | TIMES2

#Compile and test
f = MODEL.compile()
print '2 times 2: ' + str(f(2))
#2 times 2: 4.0
```

### 4.2. Connecting Models

With NNBlocks, the models can be combined in any way wanted. Consider the architecture presented in Figure 4, where an input `X` is fed to a Multilayer Perceptron, then the results are fed to a `MeanSquareError` Model that computes the mean square error between the results and another input `Y`. Such architecture can be easily achieved with few lines of code just by using Horizontal Joins, where a model's outputs are connected directly in another model's input, and Vertical Joins, where a model's output is split in two or two model's outputs are joined in the next model. The exact way these models connect can be viewed at the framework's website.

```
import nnb

#Declare blocks
X = nnb.InputLayer(ndim=1, name='X')
L1 = nnb.PerceptronLayer(insize=INP_SIZE,
                         outsize=HID1_SIZE)
L2 = nnb.PerceptronLayer(insize=HID1_SIZE,
                         outsize=HID2_SIZE)
L3 = nnb.PerceptronLayer(insize=HID2_SIZE,
                         outsize=HID3_SIZE)
Y = nnb.InputLayer(ndim=1, name='Y')
E = nnb.MeanSquareError()
```

```
#Connect blocks
NN = X | L1 | L2 | L3
COST = (NN & Y) | E
```

## 4.3. Training

After building the architecture, training can take place using the `TrainSupervisor` and one of NNBlocks' out-of-the-box trainers, some of them being:

- SGD trainer
- Adagrad trainer
- Momentum-based SGD

Training can also be customizable with custom evaluation metrics and custom procedures that can plot, adjust training parameters at run-time and set custom stopping conditions. The following example continues where the previous left off.

```
TRAINER = nnb.SGDTrainer(model=COST,
                         learning_rate=0.1)
SUP = nnb.TrainSupervisor(trainer=TRAINER,
                         plot=True,
                         dataset=DATASET,
                     eval_dataset=EV_DATASET,
                         batch_size=20,
                         epochs_num=100)
SUP.train()
```

## 4.4. Models Inside Models

As noted above, the Recurrent and Recursive NNs are a special kind of NN Model. This is because they use another Model to perform every step of the recurrence / composition. That gives a lot of flexibility to the models.
Suppose you want the recurrence of a Recurrent NN to be a simple multilayer perceptron that takes the previous output and the current word vector concatenated as input.[6]
Building this recurrence function and passing to the Recurrent NN is as simple as building any other architecture in NNBlocks.
These Models inside the recurrence / composition can have every feature an outer Model has, as they are no different than any other Model.

## 4.5. Computational Linguistic Utilities

Since NNBlocks is focused on computational linguistics neural models, there are some utilities that comes with it:

- A `WordVecsHelper` to read word vectors from a file or initialize them randomly from a tokenized text.
- A PennTreeBank file format parser.
- A PennTreeBank tree node object that extracts features from parsed sentences, so they can be fed in a Recursive NN.

NNBlocks is still in development and more tools should be added in the future.

[6]This recurrence could be way more complicated, but for the sake of the example let's leave it at that.

## 5. Conclusion

Now that the framework is presented, the implementation of real models can be discussed. NNBlocks takes advantage of modular behaviour of NNs architectures. So implementing models with NNBlocks is a matter of finding these modular behaviours.
In the CBOW model, we can identify three simple modular operations:

1. Averaging of the input vectors – This is not an out-of-the-box Model in NNBlocks, but as seen in Section 4.1. this Model can be easily built with few lines of code.
2. Doing a feedforward operation in a perceptron layer
3. Feeding the result to a Softmax layer

So it is clear that building such an architecture is easy with NNBlocks.
Building a Recursive or Recurrent NN with NNBlocks is just a matter of choosing the recurrence / composition function. Again, these functions, represented by Models, can also have their own modularity inside them. This gives a great flexibility for building.
As we can see, NNBlocks can be a great tool to build and train computational linguistics neural models.

## 6. Bibliographical References

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June. Oral Presentation.

Goller, C. and Kuchler, A. (1996). Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Le, P. and Zuidema, W. (2015). Compositional distributional semantics with long short term memory. *arXiv preprint arXiv:1503.02510.*

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781.*

Mikolov, T. (2012). Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April.*

Socher, R., Huang, E. H., Pennin, J., Manning, C. D., and Ng, A. Y. (2011). Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. (2012). Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1201–1211. Association for Computational Linguistics.

Socher, R., Bauer, J., Manning, C. D., and Ng, A. Y. (2013a). Parsing with compositional vector grammars. In *In Proceedings of the ACL conference*. Citeseer.

Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013b). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer.