DEEP NEURAL NETWORKS FROM A PRACTICAL CATEGORY THEORY POINT OF VIEW

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Abstract. 1. Summary

I make the case to view Deep Neural Networks (DNN) from a practical Category Point of view, namely to consider the algebraic properties of tensors and R-modules and how the elements of a DNN are represented by such mathematical objects. I do this to place the mathematical formulation of DNNs as general (and rigorous) as possible. I also do this, for practical reasons, to provide a concrete software design pattern, in object-oriented programming (OOP), in implementing DNNs. There should be a 1-to-1 correspondence, isomorphism, between a DNN's mathematical description and OOP design pattern, so to have a universally agreed upon OOP hierarchy.

Finally, I provide a working implementation of DNNs in Python, in both `theano` and `tensorflow` frameworks.

For an introduction to Neural Networks (NN) or i.e. Deep Neural Networks, I will reference the Coursera "MOOC" (Massive Open Online Course) by Prof. Andrew Ng, "Machine Learning Introduction."[1]

First, I will expound upon the rationale for representing input data as right R-modules.

Date: 14 juiellet 2017.

Key words and phrases. Machine Learning, Neural Networks, Deep Neural Networks, Category Theory, R-Modules, CUDA C/C++, theano, tensorflow.

2. Input Data as Right R-modules

Consider, as a start, the total given (training) input data, consisting of $m \in \mathbb{Z}^+$ (training) examples, each example, say the ith example, being represented by a "feature" vector of d features, $X^{(i)} \in \mathbb{K}^d$, where $\mathbb{K}$ is a field or (categorical) classes, i.e. as examples of fields, the real numbers $\mathbb{R}$, or integers $\mathbb{Z}$, so that $\mathbb{K} = \mathbb{R}, \mathbb{Z}$ or $\mathbb{K} = \{0, 1, \dots K-1\}$, where K is the total number of classes that a feature could fall into (for "categorical data"). Note that for this case, the case of $\mathbb{K} = \{0, 1, \dots K-1\}$, for K classes, though labeled by integers, this set of integer labels is *not* equipped with ordered field properties (it is meaningless to say $0 < 1$, for example), nor the usual field (arithmetic) operations (you cannot add, subtract, multiply, or even take the modulus of these integers).

How can we "feed into" our machine such (categorical) class data? Possibly, we should intuitively think of the Kronecker Delta function:

$$\delta_{iJ} = \begin{cases} 0 & \text{if } i \neq J \\ 1 & \text{if } i = J \end{cases}$$

for some (specific) class J, represented by an integer. So perhaps our machine can learn kronecker delta, or "signal"-like functions that will be "activated" if the integer value of a piece (feature) of data is exactly equal to J and 0 otherwise. This is essentially what the so-called "one-hot encoding" does in turning "categorical data" (some finite set) into a numerical value (a vector of 0s and 1s of dimension that is equal to the number of classes).

Onward, supposing $\mathbb{K}$ is a field, consider the total given input data of m examples:

$$\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$$

One can arrange such input data into a $m \times d$ matrix. We want to do this, for one reason, to *take advantage of the parallelism afforded by GPU(s)*. Thus we'd want to act upon the entire input data set $\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$.

We'd also want to, in the specific case of using the `theano` framework, do *parallel reduce* in order to do a *summation*, $\sum_{i=1}^m$, over all (training) examples, to obtain a cost function(al) J [2].

For `theano`, parallel `reduce` and `scan` operations can only be done over the first (size) dimension of a `theano` tensor[2]. Thus, we write the total input data as such:

$$\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m \mapsto X_j^{(i)} \in \mathrm{Mat}_{\mathbb{K}}(m, d) \tag{1}$$

i.e. $X_j^{(i)}$ is a $m \times d$ matrix of $\mathbb{K}$ values, with each ith row corresponding to the $i = 1, 2, \dots m$th example, and jth column corresponding to the $j = 1, 2, \dots d$th feature (of the feature vector $X^{(i)} \in \mathbb{K}^d$).

Let's, further make the following abstraction in that the input data $\{X^{(i)} \in \mathbb{K}^d\}_{i=1,2,\dots m}^m$ is really an element of a right R-module $\mathbf{X}$, in the category of right R-modules $\mathbf{Mod}_R$ with ring R, R not necessarily being commutative (see Rotman (2010) [3] for an introduction on non-commutative R-modules).

A reason for this abstraction is that if we allow the underlying ring R to be a field $\mathbb{K}$, (e.g. $\mathbb{K} = \mathbb{R}, \mathbb{Z}$), then the "usual" scalar multiplication by scalars is recovered. But we also need to equip $X \in \mathbf{Mod}_R$ with a *right action*, where ring R is *noncommutative*, namely

$$R = \mathrm{Mat}_{\mathbb{K}}(d, s) \cong L(\mathbb{K}^d, \mathbb{K}^s)$$

where $\mathrm{Mat}_{\mathbb{K}}(d,s)$ denotes the ring of all matrices over field $\mathbb{K}$ of matrix (size) dimensions $d \times s$ (it has d rows and s columns), $\cong$ is an isomorphism, $L(\mathbb{K}^d, \mathbb{K}^s)$ is the space of all (linear) maps from $\mathbb{K}^d$ to $\mathbb{K}^s$. If $\mathbb{K}$ is a field, this isomorphism exists.

Thus, for

$$\begin{aligned} & X \in \mathbf{X} \in \mathrm{Mod}_R \\ & R = \mathrm{Mat}_{\mathbb{K}}(d,s) \cong L(\mathbb{K}^d, \mathbb{K}^s) \end{aligned} \tag{2}$$

3. Introducing Θ, representing the "weights" of a DNN, as a mapping (homomorphism) between R-modules

Let $\Theta \in R$. Θ is also known as the "parameters" or "weights" (and is denoted by w or W by others).

Consider, as a first (pedagogical) step, only a single example ($m = 1$). X is only a single feature vector, $X \in \mathbb{K}^d$. Then for basis $\{e_\mu\}_{\mu=1\dots d}$ of $\mathbb{K}^d$, corresponding dual basis $\{e^\mu\}_{\mu=1\dots d}$ (which is a basis for dual space $(\mathbb{K}^d)^*$), then

$$\begin{gathered} X\Theta = X^\mu e_\mu ({\Theta_\nu}^j e^\nu \otimes e_j) = \\ X^\mu {\Theta_\nu}^j e^\nu(e_\mu) \otimes e_j = X^\mu {\Theta_\nu}^j \delta^\nu_\mu = \\ e_j = X^\mu {\Theta_\mu}^j e_j \text{ where} \\ \mu, \nu = 1 \dots d \\ j = 1 \dots s \end{gathered}$$

In this case where X is simply a vector, one could think of X as a "row matrix" and Θ is a matrix, acting on the right, in matrix multiplication.

One should note that while it is deceptively simple to consider tensors as the Cartesian product, $(\mathbb{K}^d)^* \times \mathbb{K}^s$, i.e. that in general,

$$(\mathbb{K}^d)^* \times \mathbb{K}^s \neq T^{(d,s)}$$

where $T^{(d,s)}$ is the set of all tensors of rank (d,s) (depending upon so-called co-variant, contravariant convention). Rather, one should take the quotient of this "raw" Cartesian product with the equivalence relation of a particular submodule (which essentially defines what the zero equivalence class is; cf. Jeffrey Lee (2009) [4], Conlon (2008) [5]). It would be interesting to verify this point in terms of the needs for a tensor representation of the "weights" of a DNN.

Now suppose, in general, $X \in \mathbf{X} \in \mathbf{Mod}_R$, where X could be a $m \times d$ matrix, or higher-dimensional tensor. For a concrete example, say $\mathbf{X} = \mathrm{Mat}_{\mathbb{K}}(m,d)$. We not only have to equip this right R-module with the usual scalar multiplication, setting ring $R = \mathbb{K}$, but also the right action version of matrix multiplication, so that $R = \mathrm{Mat}_{\mathbb{K}}(d,s)$. This R is *non-commutative*, thus, necessitating the abstraction to right R-modules.

Also, from a practical point of view, we'd like our model to take an arbitrary number of samples/examples, $m \in \mathbb{Z}^+$. To encode this, for example in `tensorflow`, this is done by setting the so-called `shape` parameter to have a value of `None` - the shape will be finally specified when a `tf.Session` is instantiated and the computational graph is run with class method `.run`. But the mapping or computation must proceed without knowing exactly the value of m, and so the weights and bias (Θ, b) act *on the right*.

Indeed, for

$$\Theta \in L(\mathrm{Mat}_{\mathbb{K}}(m,d), \mathrm{Mat}_{\mathbb{K}}(m,s)) \cong (\mathrm{Mat}_{\mathbb{K}}(m,d))^* \otimes \mathrm{Mat}_{\mathbb{K}}(m,s) \cong \mathrm{Mat}_{\mathbb{K}}(d,s), \text{ and so}$$
$$X\Theta \in \mathrm{Mat}_{\mathbb{K}}(m,s)$$

Further

$$X\Theta \in \mathrm{Mat}_{\mathbb{K}}(m,s) \in \mathbf{Mod}_R$$

with ring R in this case being $R = \mathrm{Mat}_{\mathbb{K}}(s,s_2) \cong L(\mathbb{K}^s, \mathbb{K}^{s_2})$.

Since $X\Theta$ is an element in a R-module, it is an element in an (additive) abelian group. We can add the "intercept vector" b (in theano, it'd be the usual theano vector, but with its dimensions "broadcasted" for all m examples, i.e. for all m rows).

$$X\Theta + b \in \mathrm{Mat}_{\mathbb{K}}(m,s)$$

Considering these 2 operations on X, the "matrix multiplication on the right" or right action Θ, and addition by b together, through *composition*, (Θ, b), we essentially have

$$\boxed{X \in \mathbf{X} \in \mathbf{Mod}_{R_1} \xrightarrow{(\Theta,b)} X\Theta + b \in \mathbf{X_2} \in \mathbf{Mod}_{R_2}} \tag{3}$$

where

$$R_1 = \mathrm{Mat}_{\mathbb{K}}(d,s_1) \cong L(\mathbb{K}^d, \mathbb{K}^{s_1})$$
$$R_2 = \mathrm{Mat}_{\mathbb{K}}(s_1,s_2) \cong L(\mathbb{K}^{s_1}, \mathbb{K}^{s_2})$$

4. Axons: what matters is not the specific nature of objects alone, but the relationships between objects

Consider a(n artificial) neural network (NN) of $L+1 \in \mathbb{Z}^+$ "layers" representing $L+1$ neurons, with each layer or neuron represented by a vector $a^{(l)} \in \mathbb{K}^{s_l}$, $s_l \in \mathbb{Z}^+$, $l = 1, 2, \dots L+1$ (or, counting from 0, $l = 0, 1, \dots L$, so-called 0-based counting). Again, $\mathbb{K}$ is either a field (e.g. $\mathbb{K} = \mathbb{R}, \mathbb{Z}$), or categorical classes (which is a subset of $\mathbb{Z}^+$, but without any field properties, or field operations).

Recall the usual (familar) NN, accepting that we do right action multiplications (matrices act on the right, vectors are represented by "row vectors", which, actually, correspond 1-to-1 with `numpy`/`theano`/`tensorflow` arrays, exactly, in that 1-dimensional arrays in `numpy`/`theano`/`tensorflow` default to being treated as row vectors).

Recall also that the sigmoidal or (general) *activation* function, $\psi^{(l)}$, acts element-wise on a vector, matrix, or tensor. And so

$$\psi^{(l)} : \mathbb{K}^d \to \mathbb{K}^d$$
$$\psi^{(l)} : a_i^{(l)} \mapsto \psi^{(l)}(a_i^{(l)}) \qquad \forall\, i = 0, 1, \dots d-1$$
$$\psi^{(l)} : \mathrm{Mat}_{\mathbb{K}}(d,s) \to \mathrm{Mat}_{\mathbb{K}}(d,s)$$
$$\psi^{(l)} : X_{ij}^{(l)} \to \psi^{(l)}(X_{ij}^{(l)})$$

To generalize this, $\psi^{(l)}$ maps an R-module in Mod_R to another R-module, but belonging in the same domain:

$$\psi^{(l)} : \mathrm{Mod}_R \to \mathrm{Mod}_R$$

and its operation on each "entry" of a R-module is clear.

The lth layer was computed from the "last" or "previous" layer, in a sequence of layers, as computed as follows:

$$\begin{aligned} z^{(l)} &:= a^{(l-1)}\Theta^{(l)} + b^{(l)} \\ a^{(l)} &:= \psi^{(l)}(z^{(l)}) \end{aligned} \tag{4}$$

where $\Theta^{(l)}, b^{(l)}$ is as above, except there will be a total of L of these tuples ($l = 1, 2, ...L$).

With $(\Theta^{(l)}, b^{(l)})$ representing the (right action) linear transformation

$$(\Theta^{(l)}, b^{(l)})(a^{(l-1)}) := a^{(l)}\Theta^{(l)} + b^{(l)} \tag{5}$$

essentially,

$$\begin{aligned} \mathbf{Mod}_{R^{(l-1)}} &\xrightarrow{(\Theta^{(l)}, b^{(l)})} \mathbf{Mod}_{R^{(l)}} \xrightarrow{\psi^{(l)}\odot} \mathbf{Mod}_{\mathbb{R}^{(l)}} \\ (\mathbb{K}^{s_{l-1}})^m &\xrightarrow{(\Theta^{(l)}, b^{(l)})} (\mathbb{R}^{s_l})^m \xrightarrow{\psi^{(l)}\odot} (\mathbb{K}^{s_l})^m \\ a^{(l-1)} &\xmapsto{(\Theta^{(l)}, b^{(l)})} z^{(l)} \xmapsto{\psi^{(l)}\odot} a^{(l)} \end{aligned} \tag{6}$$

Since we need to operate with the activation function $\psi^{(l)}\odot$ elementwise, we (implicitly) equip $\mathbf{Mod}_{R^{(l+1)}}$ with the Hadamard product. In fact, with composition, we can represent the above mapping between the $l-1$ and lth layers as

$$\mathbf{Mod}_{R^{(l-1)}} \xrightarrow{\psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} \mathbf{Mod}_{\mathbb{R}^{(l)}}$$

$$(\mathbb{K}^{s_{l-1}})^m \xrightarrow{\psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} (\mathbb{K}^{s_l})^m$$

$$a^{(l-1)} \xmapsto{\psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} a^{(l)} \tag{7}$$

The lesson (and the whole point of category theory) is this: instead of thinking of layers, each separately, think of or focus on the relationship, the relations, between each layers, as one whole entity. Indeed, the historical development, and its defining characteristic, of category theory is that what does not matter is the specific nature of the mathematical objects by themselves, but the general relationship between the mathematical objects. Thus, I propose considering the whole of the mapping shown in Eq. 7 an **axon**. The lth axon, for $l = 1, 2, \dots L$, is a map between the $(l-1)$th layer and the lth layer. There are L total axons in a NN, with $L+1$ layers (which include an input and output layer).

To reiterate, the axon consists of the map $\psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})$, and the spaces it maps between, $\mathbf{Mod}_{R^{(l-1)}}$ to $\mathbf{Mod}_{R^{(l)}}$.

5. Deep Neural Networks (DNN) from the Category Point of View

Suppose we "feed in" input data X into the first or 0th layer of this NN. This means that for $a^{(0)} \in \mathbb{R}^d$,

$$a^{(0)} = X^{(i)}$$

for the ith (training) example.

The "output" layer, layer L, should output the *predicted* value, given X. So

$$a^{(L)} \in \mathbb{R} \text{ or } \{0, 1, \dots K-1\} \text{ or } [0, 1]$$

for regression, or classification (so it takes on discrete values) or the probability likelihood of being in some class k, respectively.

The entire DNN can mathematically expressed as follows:

(8)

$$\begin{array}{ccc} \mathbf{Mod}_{R^{(0)}} & \xrightarrow{\prod_{l=1}^{L} \psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} & \mathbf{Mod}_{R^{(L)}} \\ (\mathbb{K}^d)^m & \xrightarrow{\prod_{l=1}^{L} \psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} & (\mathbb{K}^{s_L} \text{ or } \mathbb{R} \text{ or } \{0, 1, \dots K-1\} \text{ or } [0, 1])^m \\ X & \xmapsto{\prod_{l=1}^{L} \psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})} & a^{(L)} = \widehat{y} \end{array}$$

with $\widehat{y}$ being notation for the target output that the DNN predicts, given input data X. Then the (small) *commutative diagram* shown in Eq. 8 compactly and succinctly tells us what DNNs do: given $2L$ R-modules, $(\Theta^{(l)}, b^{(l)}$, a choice of L *activation* functions (element-wise mappings) $\psi^{(l)}$, and its sequential composition, then given input data X, which in general is a R-module (it can be a single value, a (row) vector, matrix, tensor, etc.), it will map X to a prediction $\widehat{y}$.

Referencing other names for what is essentially Eq. 8, this is essentially a *feed-forward* network, in that given the input data X, it is "fed-forward" through the application of a sequence of $\psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})$ mappings to yield the prediction $\widehat{y}$.

6. Cost functional J

We need to calculate a cost functional J (rather than a cost function) for this DNN. Consider the training process. The input training dataset and corresponding target data to train on doesn't change (usually), X,y, respectively, and so given X,y, J is a function of weights and bias $(\mathbf{\Theta}, \mathbf{b}) \equiv (\Theta^{(1)}, b^{(1)}), \dots (\Theta^{(L)}, b^{(L)})$ (where I will use this notation short cut defined here). But taking J to be dependent upon X, y themselves, then J would be a *cost functional*:

(9)
$$J_{(X,y)} : (L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L \to C^k(L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L$$
$$J_{(X,y)} : (\mathbf{\Theta}, \mathbf{b}) \mapsto J_{(X,y)}(\mathbf{\Theta}, \mathbf{b})$$
$$J : \mathrm{Mod}_R \times \mathrm{Mod}_R \times (L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L \to L((L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L, \mathbb{R})$$
$$J : (X, y), (\mathbf{\Theta}, \mathbf{b}) \mapsto J_{(X,y)}$$

In this way, we can define a *gradient*, grad, of J with respect to $(\mathbf{\Theta}, \mathbf{b})$, consisting of partial derivatives with respect to the components or entries of $(\mathbf{\Theta}, \mathbf{b})$,

$$\mathrm{grad}J = \mathrm{grad}_{(\mathbf{\Theta},\mathbf{b})}J = \frac{\partial J}{\partial \Theta_I^{(l)}}, \frac{\partial J}{\partial b_H^{(m)}}$$

where I, H are generalized indices that could be *multi-indices* (e.g. $I = ij$, with i being the "row" index, and j being the "column" index), and $l, m = 1, 2, \dots L$.

6.1. **Training on J and X, y, and the DNN model.** The goal is to minimize J, with respect to $(\mathbf{\Theta}, \mathbf{b}) \in (L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L$. It is unclear whether critical points do exist for this space. Nevertheless, one then trains a DNN through some iterative process that minimizes J, changing $(\mathbf{\Theta}, \mathbf{b})$, whether through gradient descent, Adagrad, conjugate gradient descent, etc. But a common characteristic of training is that we take given input data and target data we train on, $X, y \in \mathrm{Mod}_R$, a cost functional J, and return a model, $(\mathbf{\Theta}, \mathbf{b})$:

$$X, y \in \mathrm{Mod}_R \xmapsto{\text{train}} (\mathbf{\Theta}, \mathbf{b}) \in (L(\mathrm{Mod}_R, \mathrm{Mod}_R) \times \mathrm{Mod}_R)^L$$

Note that this procedure is also called "backpropagation" but it is to simply run an iterative procedure to minimize J.

Thus, a *deep neural network (DNN)* model would include the following:

- Eq. 8, which includes *axons*:
  - $((\mathbf{\Theta^{(l)}}, \mathbf{b^{(l)}}), \psi^{(l)})_{l=1,\dots L}$ and $(a^{(l)})_{l=0,1,\dots L}$, i.e. L *axons*
  - the *dimensions* (which I call size dimensions) of the spaces being mapped from and into, i.e.

(10)
$$(d, s_1, s_2, \dots s_L) \in (\mathbb{Z}^+)^{L+1}$$

    with d representing the number of features, and s_L being equal to the dimensions of the target output.
- cost functional J, and input (and target) data X, y
- some training (iterative) algorithm *train*

I would argue that this constitute the entirety of the DNN model.

7. 1-to-1 correspondence (isomorphism) between this mathematical formulation and software engineering design/Object-Oriented Programming (OOP)

This 1-to-1 correspondence was inspired by the "Sage Category Framework", which was laid out in the abstract and manual for the open-source Sage Math software [6]. In there, the Sage Math development team proposed a general philosophy, and a rationale for it, in that "Building mathematical information into the system yields more expressive, more conceptual and, at the end, easier to maintain and faster code." Essentially, the hierarchy of mathematical objects and mappings

(which includes functions, functionals, homomorphisms) defined by category theory should directly translate to the structure of classes, class members and class methods, in software.

I have provided implementations of DNNs in both the `theano`[2] and `tensorflow`[7] on `github`, namely in the subdirectory `ML/` of the repository `ernestyalumni:MLgrabbag`, in files DNN.py and DNN˙tf.py, respectively.

The DNN model outlined in Subsection 6.1 correspond directly to the Python classes defined in those implementations:

- `class Axon` is $(\mathbf{\Theta^{(l)}}, \mathbf{b^{(l)}}), \psi^{(l)}$ for some $l = 1, 2, \dots L$, including the (size) dimensions of the spaces it maps between, i.e. $s_{l-1} = \text{dimMod}_{R^{l-1}}$ and $s_l = \text{dimMod}_{R^l}$
- `class Feedforward` is the diagram in Eq. 8. In both classes `Axon` and —Feedforward—, there is a `connect_through` method which corresponds directly to actually applying the mapping laid out by $\psi^{(l)} \odot (\mathbf{\Theta^{(l)}}, \mathbf{b^{(l)}})$ and $\prod_{l=1}^{L} \psi^{(l)} \odot (\Theta^{(l)}, b^{(l)})$, respectively.
- `class DNN` is what is outlined for the DNN model in Subsection 6.1, allowing for
  - the construction of different J's for your model, J where the error is a L^2 norm, J for a cross-entropy cost for logistic regression, J that includes a regularization term for each, etc. as a class method
  - some training algorithm, executed in class method `train_model`
  - class members to put the input (and target) data $X_{\text{train}}, y_{\text{train}}$ in to train the model upon

8. Final Thoughts

Certainly, there are other frameworks built on top of the frameworks provided by `theano` and `tensorflow`, namely `keras` (`https://keras.io/`) and in some higher level classes in `tensorflow` itself, namely `layers`, that would implement such a DNN model. I wanted to present here a general mathematical description of a DNN (or i.e. ANN) from the viewpoint of category theory (which is about as abstract and general as one can get in pure mathematics). The point of category theory is that the relations between mathematical objects is important, not the specific nature of the objects themselves. Thus, I introduced the notion of *axons* as opposed to layers, and build DNNs out of axons.

Again, there are certainly other well-developed frameworks such as `keras`, but I would submit, from a practical standpoint, that by understanding this mathematical prescription, one can deploy new types of NNs quickly. For instance, Jozefowicz, Zaremba, and Sutskever (2015)[8] evaluated different variations or "mutations" of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks (these are examples of Recurrent Neural Networks (RNN); these can be implemented quickly by simply applying various compositions of weights and bias, $(\mathbf{\Theta}, \mathbf{b})$ and so reusing this `Thetab` class or `Axon` class, instantiated several times, as I've shown in GRUs˙Right.py.

References

[1] Andrew Ng. Machine Learning. coursera
[2] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions".

[3] Joseph J. Rotman, **Advanced Modern Algebra** (Graduate Studies in Mathematics) 2nd Edition, American Mathematical Society; 2 edition (August 10, 2010), ISBN-13: 978-0821847411

[4] Jeffrey M. Lee. **Manifolds and Differential Geometry**, *Graduate Studies in Mathematics* Volume: 107, American Mathematical Society, 2009. ISBN-13: 978-0-8218-4815-9

[5] Lawrence Conlon. **Differentiable Manifolds** (Modern Birkhäuser Classics). 2nd Edition. Birkhäuser; 2nd edition (October 10, 2008). ISBN-13: 978-0817647667

[6] The Sage Development Team. Sage Reference Manual: Category Framework. Release 7.6. Mar. 25, 2017.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo,Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis,Jeffrey Dean,Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia,Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster,Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens,Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker,Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas,Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke,Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[8] Rafal Jozefowicz, Wojciech Zaremba, Ilya Sutskever. "An Empirical Exploration of Recurrent Network Architectures." *Proceedings of the 32nd International Conference on Machine Learning*, Lille, France, 2015. JMLR: W& CP volume 37. http://www.jmlr.org/proceedings/papers/v37/jozefowicz15.pdf