numpydl Documentation

Release 0.1.0

NumpyDL

May 11, 2017

Contents

NumpyDL is a simple deep learning library based on pure Python/Numpy. NumpyDL is a work in progress, input is welcome. The project is on GitHub.

The main features of NumpyDL are as follows:

- *Pure* in Numpy and *native* to Python
- Support basic *automatic differentiation*
- Support *commonly used models*, such as MLP, RNNs, GRUs, LSTMs and CNNs
- *Perfect documents* and easy to learn deep learning knowledge
- Flexible network configurations and learning algorithms.
- API like Keras deep learning library

The design of NumpyDL is governed by several principles:

- **Simplicity**: Be easy to use, easy to understand and easy to extend, to facilitate use in research. Interfaces should be kept small, with as few classes and methods as possible. Every added abstraction and feature should be carefully scrutinized, to determine whether the added complexity is justified.
- **Transparency**: Native to Numpy, directly process and return Python/Numpy data types. Do not rely on the functionality of Theano, Tensorflow or any such deep learning frameworks.
- **Modularity**: Allow all parts (layers, regularizers, optimizers, ...) to be used independently of NumpyDL. Make it easy to use components in isolation or in conjunction with other frameworks.
- **Focus**: "Do one thing and do it well". Do not try to provide a library for everything to do with deep learning.

CHAPTER 1

User Guides

The NumpyDL user guide explains how to install NumpyDL, how to build and train neural networks using NumpyDL, and how to contribute to the library as a developer.

Installation

NumpyDL has a couple of prerequisites that need to be installed first, but it is not very picky about versions. The most important package is Numpy. At the same time, you should install some other useful packages, such as scipy and scikit-learn. Most importantly, these packages are not required to install the specific version to fit the version of NumpyDL you choose to install.

We strongly recommend you to install the Miniconda or a bigger installer Anaconda which is a leading open data science platform powered by Python and well integrated the efficient scientific computing platform MKL.

1. Prerequisites

1.1. Python + pip

NumpyDL currently requires Python 3.3 or higher to run. Please install Python via the package manager of your operating system if it is not included already.

Python includes `pip` for installing additional modules that are not shipped with your operating system, or shipped in an old version, and we will make use of it below. We recommend installing these modules into your home directory via `--user`, or into a virtual environment via `virtualenv`.

1.2. C compiler

Numpy/scipy require a C compiler if you install them via `pip`. On Linux, the default compiler is usually‘‘gcc‘‘, and on Mac OS, it’s `clang`. On Windows, we recommend you to install the Miniconda or Anaconda. Again, please install them via the package manager of your operating system.

1.3. numpy/scipy + BLAS

NumpyDL requires numpy of version 1.6.2 or above, and sometimes also requires scipy 0.11 or above. Numpy/scipy rely on a BLAS library to provide fast linear algebra routines. They will work fine without one, but a lot slower, so it is worth getting this right (but this is less important if you plan to use a GPU).

If you install numpy and scipy via your operating system's package manager, they should link to the BLAS library installed in your system. If you install numpy and scipy via `pip install numpy` and `pip install scipy`, make sure to have development headers for your BLAS library installed (e.g., the `libopenblas-dev` package on Debian/Ubuntu) while running the installation command. Please refer to the numpy/scipy build instructions if in doubt.

2. Stable NumpyDL release

To install a version that is known to work, run the following command:

```
pip install -r https://github.com/oujago/NumpyDL/blob/master/requirements.txt
```

```
pip install npdl
```

If you do not use `virtualenv`, add `--user` to both commands to install into your home directory instead. To upgrade from an earlier installation, add `--upgrade`.

3. Development installation

Alternatively, you can install NumpyDL from source, in a way that any changes to your local copy of the source tree take effect without requiring a reinstall. This is often referred to as *editable* or *development* mode. Firstly, you will need to obtain a copy of the source tree:

```
git clone https://github.com/oujago/NumpyDL.git
```

It will be cloned to a subdirectory called `NumpyDL`. Make sure to place it in some permanent location, as for an *editable* installation, Python will import the module directly from this directory and not copy over the files. Enter the directory and install the known good version of Theano:

```
cd NumpyDL
pip install -r requirements.txt
```

To install the NumpyDL package itself, in editable mode, run:

```
pip install --editable
```

As always, add `--user` to install it to your home directory instead.

Optional: If you plan to contribute to NumpyDL, you will need to fork the NumpyDL repository on GitHub. This will create a repository under your user account. Update your local clone to refer to the official repository as `upstream`, and your personal fork as `origin`:

```
git remote rename origin upstream
git remote add origin https://github.com/<your-github-name>/NumpyDL.git
```

If you set up an SSH key, use the SSH clone URL instead: `git@github.com:<your-github-name>/NumpyDL.git`.

You can now use this installation to develop features and send us pull requests on GitHub, see development!

CHAPTER 2

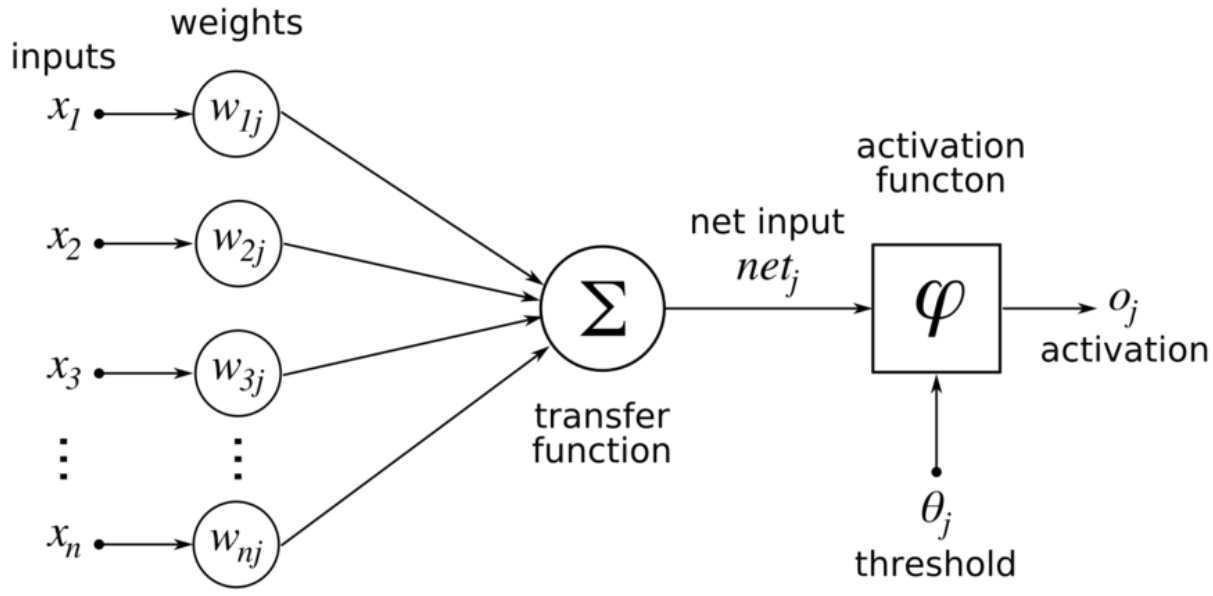
Tutorials

This is the tutorials of NumpDL.

Activation Functions

1. What is activation function?

In computational networks, the activation function of a node defines the output of that node given an input or set of inputs. In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing in the cell.



2. Why need activation function?

Neural networks compose several functions in layers: the output of a previous layer is the input to the next layer. If you compose linear functions, these functions are all linear. So the result of stacking several linear functions together is a linear function. Using a nonlinear function makes the map from the input to the output nonlinear.

For example, a *ReLU* function's output is either 0 or positive. If the unit is 0, it is effectively "off," so the inputs to the unit are not propagated forward from that function. If the unit is on, the input data is reflected in subsequent layers through that unit. ReLU itself is not linear, and neither is the composition of several layers of several ReLU functions. So the mapping from inputs to classification outcomes is not linear either.

Without activation function many layers would be equivalent to a single layer, as each layer (without an activation function) can be represented by a matrix and a product of many matrices is still a matrix:

$$M = M_1 M_2 \cdots M_n$$

3. Commonly used activation functions

Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functions you may encounter in practice:

3.1. Sigmoid

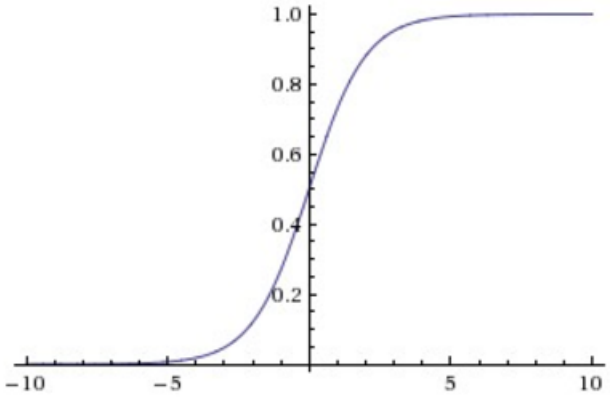


Fig. 2.1: Sigmoid non-linearity squashes real numbers to range between $[0, 1]$.

The sigmoid non-linearity has the mathematical form $(x) = 1/(1 + e^x)$ and is shown in the image above. It takes a real-valued number and "squashes" it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks:

- *Sigmoids saturate and kill gradients*. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.

- *Sigmoid outputs are not zero-centered.* This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$)), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

3.2. Tangent

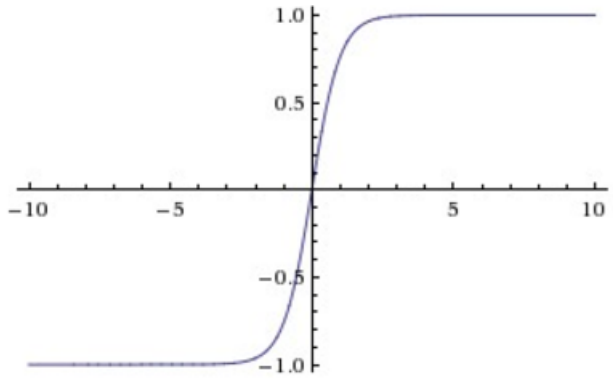


Fig. 2.2: The tanh non-linearity squashes real numbers to range between $[-1, 1]$.

The tanh non-linearity is shown on the image above. It squashes a real-valued number to the range $[-1, 1]$. Like the *sigmoid* neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds:

$$tanh(x) = 2(2x)1.$$

3.3. ReLU

The Rectified Linear Unit has become very popular in the last few years. It computes the function

$$f(x) = max(0, x)$$

In other words, the activation is simply thresholded at zero (see image above). There are several pros and cons to using the ReLUs:

- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al[1].) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to *tanh*/*sigmoid* neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

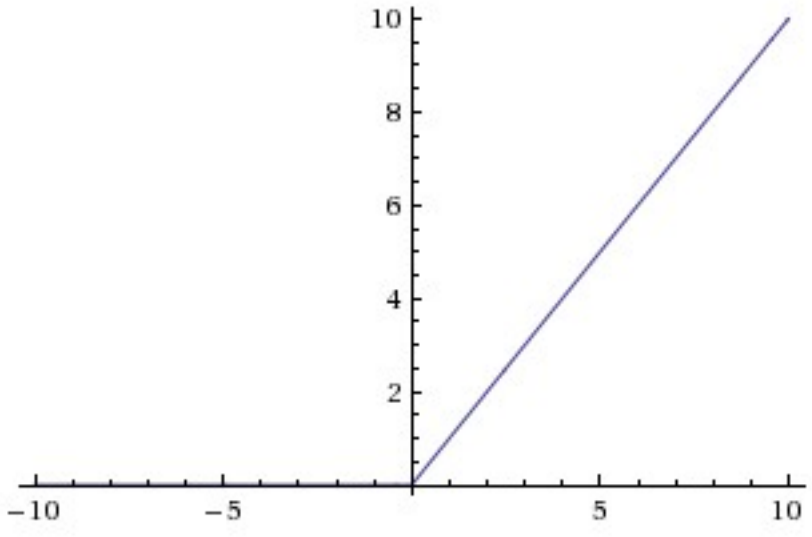


Fig. 2.3: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$.

- (-) Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

3.4. Leaky ReLU

Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes

$$f(x) = \begin{cases} x & x < 0 \\ x & x >= 0 \end{cases}$$

where is a small constant. Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons, introduced in *Delving Deep into Rectifiers, by Kaiming He et al., 2015*[2]. However, the consistency of the benefit across tasks is presently unclear.

3.5. Maxout

Other types of units have been proposed that do not have the functional form

$$f(wTx + b)$$

where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the *Maxout* neuron (introduced recently by Goodfellow et al[3].) that generalizes the ReLU and its leaky version. The

[2] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

[3] Goodfellow, Ian J., et al. "Maxout networks." arXiv preprint arXiv:1302.4389 (2013).

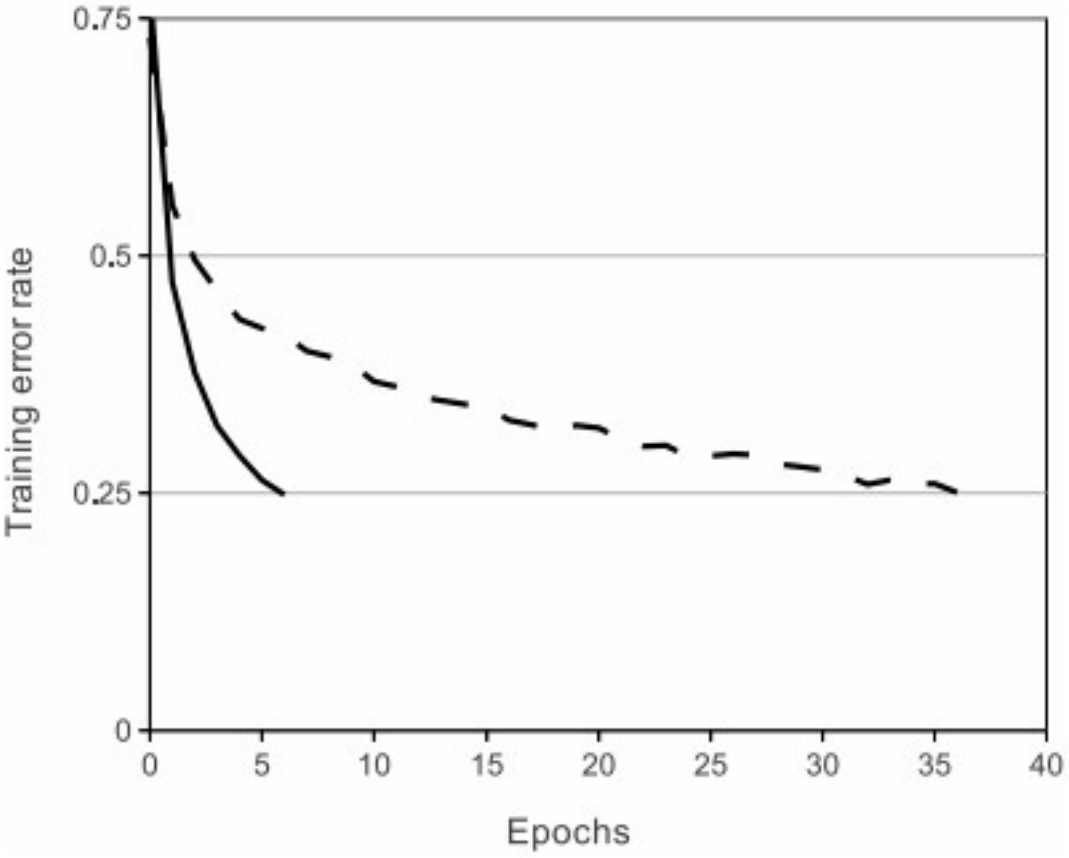
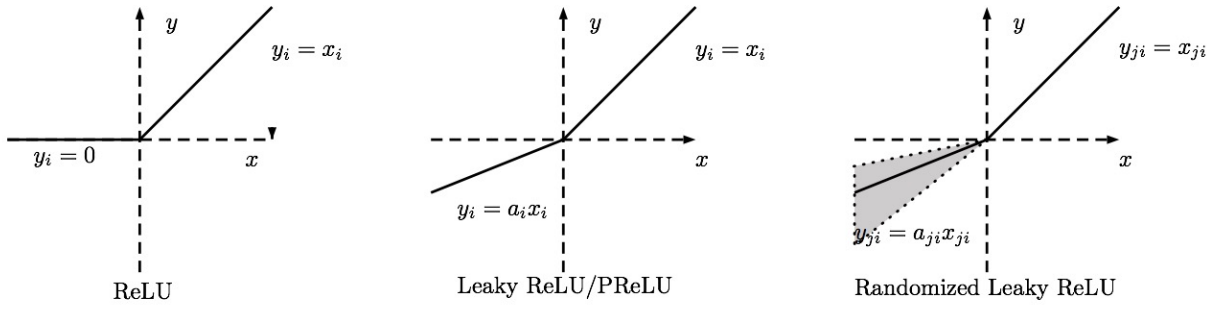


Fig. 2.4: A plot from Krizhevsky et al.[1] paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.



Maxout neuron computes the function

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

Notice that both *ReLU* and *Leaky ReLU* are a special case of this form (for example, for ReLU we have $w1, b1 = 0$). The *Maxout* neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

This concludes our discussion of the most common types of neurons and their activation functions. As a last comment, it is very rare to mix and match different types of neurons in the same network, even though there is no fundamental problem with doing so.

4. What activation should I use?

Use the *ReLU* non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give *Leaky ReLU* or *Maxout* a try. Never use *sigmoid*. Try tanh, but expect it to work worse than ReLU/Maxout.

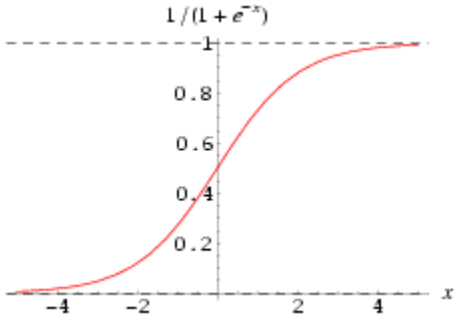
5. References

Multi-layer Perceptron MLP

1. Sigmoid function

BP algorithm is mainly due to the emergence of Sigmoid function, instead of the previous threshold function to construct neurons.

The Sigmoid function is a monotonically increasing nonlinear function. When the threshold value is large enough, the threshold function can be approximated.



The Sigmoid function is usually written in the following form:

$$f(\alpha) = \frac{1}{1+e^{-\alpha}}$$

The value range is (-1,1), which can be used instead of the neuron step function:

Due to the complexity of the network structure, the Sigmoid function is used as the transfer function of the neuron. This is the basic idea of multilayer perceptron backpropagation algorithm.

$$y = f(x) = \frac{1}{1 + e^{-\sum_{i=1}^{n} w_i x_i - w_0}}$$

2. BP

BP algorithm is the optimization of the network through the iterative weights makes the actual mapping relationship between input and output and the desired mapping, descent algorithm by adjusting the layer weights for the objective function to minimize the gradient. The sum of the squared error between the predicted output and the expected output of the network on one or all training samples

$$J(w) = \frac{1}{2}\sum_{j=1}^{S}(t_j - a_j)^2 = \frac{1}{2}\|t - a\|^2$$

$$J_{Total}(w) = \frac{1}{2}\sum_{i=1}^{N}\|t_i - a_i\|^2$$

The error of each unit is calculated by layer by layer error of output layer:

$$\Delta w_j^{(k)} = -\eta\frac{\partial J}{\partial w_j^{(k)}} = -\eta\frac{\partial J}{\partial n_j^{(k)}}\frac{\partial n_j^{(k)}}{\partial w_j^{(k)}}$$

$$= -\eta\frac{\partial J}{\partial n_j^{(k)}}a^{(k-1)} = -\eta\delta_j^{(k)}a^{(k-1)}$$

Back Propagation Net (BPN) is a kind of multilayer network which is trained by weight of nonlinear differentiable function. BP network is mainly used for:

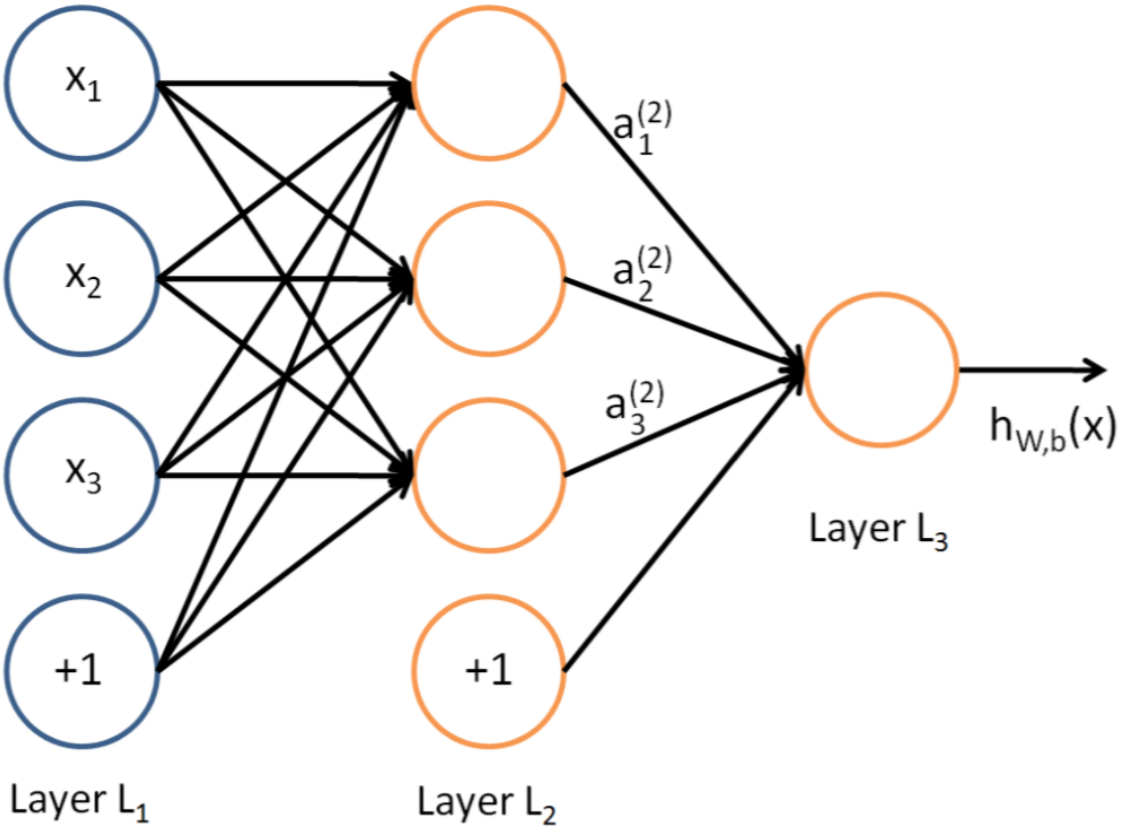
1. function approximation and prediction analysis: using the input vector and the corresponding output vector to train a network to approximate a function or to predict the unknown information;
2. pattern recognition: using a specific output vector to associate it with the input vector;
3. classification: the input vector is defined in the appropriate manner;
4. data compression: reduce the output vector dimension to facilitate transmission and storage.

For example, a three tier BPN structure is as follows:

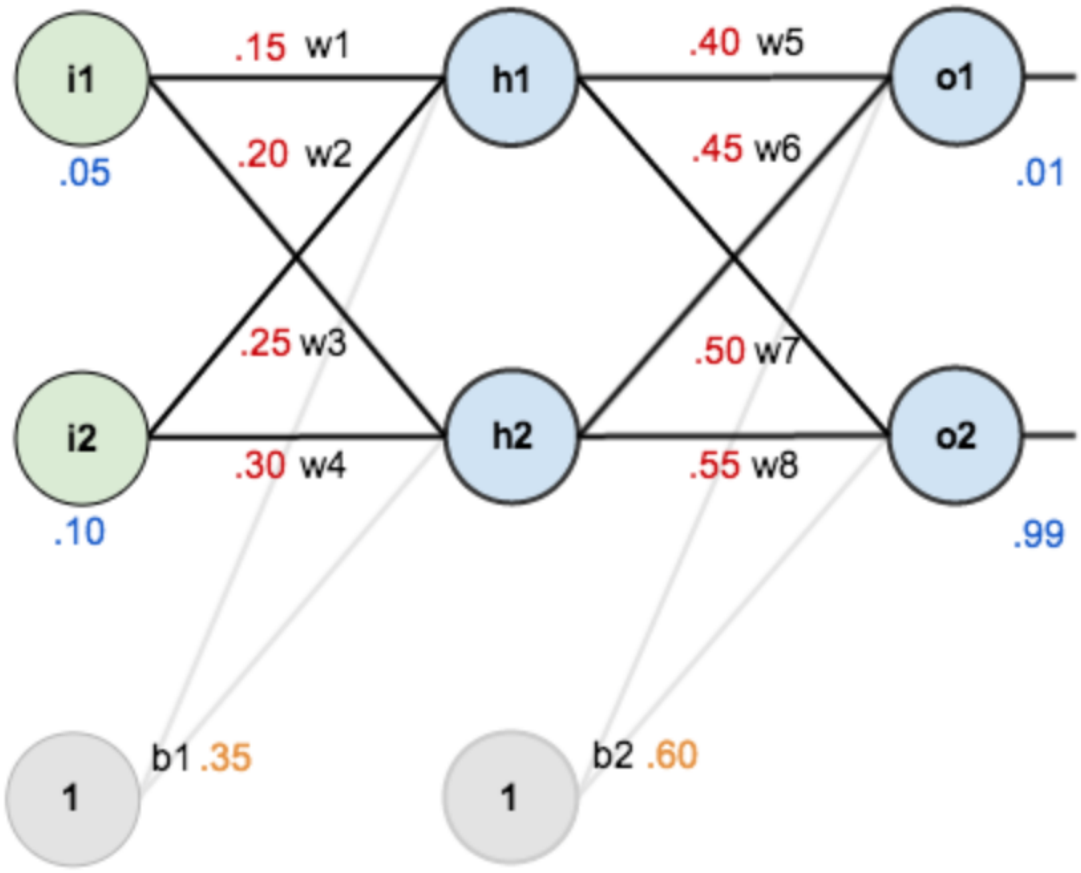
It consists of three layers: input layer, hidden layer and output layer. The unit of each layer is connected with all the units of the adjacent layer, and there is no connection between the units in the same layer. When a pair of learning samples are provided to the network, the activation value of the neuron is transmitted from the input layer to the output layer through the intermediate layers, and the input response of the network is obtained by the neurons in the output layer. Next, according to the direction of reducing the output of the target and the direction of the actual error, the weights of each link are modified from the output layer to the input layer.

3. Example

Suppose you have such a network layer:


x_1
x_2
x_3
+1
Layer L_1
+1
Layer L_2
$a_1^{(2)}$
$a_2^{(2)}$
$a_3^{(2)}$
Layer L_3
$h_{W,b}(x)$

The first layer is the input layer, two neurons containing i1, i2, b1 and intercept; the second layer is the hidden layer, including two neurons h1, h2 and intercept b2, the third layer is the output of o1, o2 and wi are each line superscript connection weights between layers, we default to the activation function sigmoid function. Now give them the initial value, as shown below:



Among them, Input data i1=0.05, i2=0.10; Output data o1=0.01, o2=0.99; Initial weight w1=0.15, w2=0.20, w3=0.25, w4=0.30;

w5=0.40, w6=0.45, w7=0.50, w8=0.88;

Objective: to give input data i1, i2 (0.05 and 0.10), so that the output is as close as possible to the original output o1, o2 (0.01 and 0.99).

3.1. Step 1 Forward Propagation

1. Input layer —-> Hidden layer:

Calculate the input weighted sum of neurons h1:

o1, the output of neuron h1: (Activation function sigmoid is required here):

Similarly, o2, the output of neuron h2 can be calculated:

2. Hidden layer —-> Output layer:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

The values of o1 and o2 in the output layer are calculated:

This propagation process is finished, we get the output value of [0.75136079, 0.772928465], and the actual value of [0.01, 0.99] far from now, we for the error back-propagation, update the weights, to calculate the output.

3.2. Step 2 Back Propagation

1. Calculate the total error

Total error: (square error)

For example, the target output for o1 is 0.01 but the neural network output 0.75136507, therefore its error is:

Repeating this process for o_2 (remembering that the target is 0.99) we get:

The total error for the neural network is the sum of these errors:

2. Hidden layer —-> Hidden layer weights update:

Take the weight parameter w5 as an example, if we want to know how much impact the w5 has on the overall error, we can use the global error to obtain the partial derivative of w5: (chain rule)

The following figure can be more intuitive to see how the error is spread back:

Now we were calculated for each value:

Calculate

Calculate

(This step is actually a derivative of the sigmoid function)

Calculate

Putting it all together:

In this way, we calculate the overall error E (total) to the w5 partial guide. Look at the above formula, we found:

$$out_{h2} = 0.596884378$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

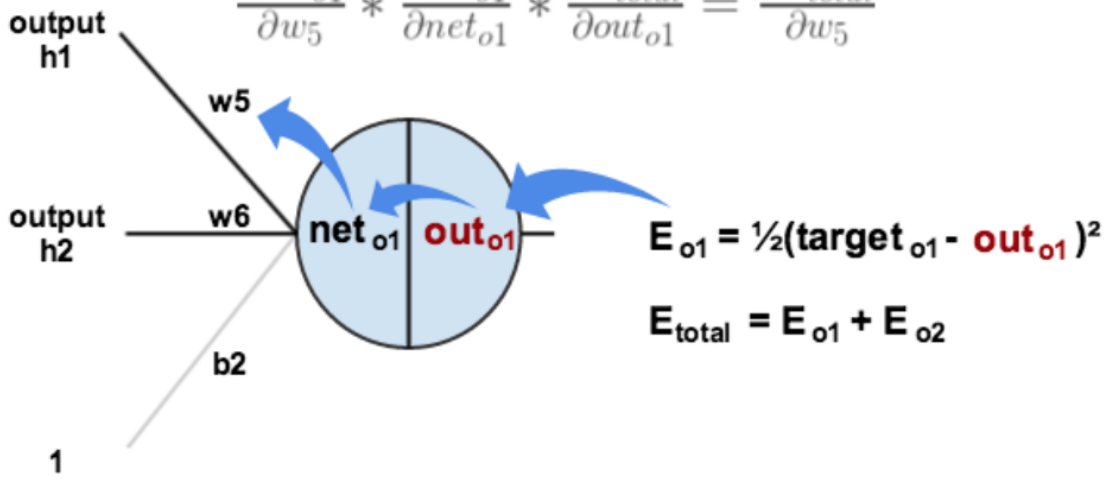
$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$\frac{\partial E_{total}}{\partial out_{o1}}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$\frac{\partial out_{o1}}{\partial net_{o1}}$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$\frac{\partial net_{o1}}{\partial w_5}$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

In order to express convenience,.. figure:: pics/27.pngis used to express the error of output layer:

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

Therefore, the overall error E (total) can be written as a partial derivative formula for w5:

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

If the output layer error meter is negative, it can also be written:

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

Finally, we update the value of w5:

(Among them, .. figure:: pics/32.pngis the learning rate, here we take 0.5)

Similarly, update w6, w7, w8:

3. Hidden layer —-> Hidden layer weights update:

In fact, with the method above said almost, but there is a need to change, calculate the total error of the above w5 guide, from out (o1) —->net (o1) —->w5, but in the hidden layer between the weight update, out (h1) —->net (h1) —->w1 and out (h1) will accept E(o1) and E(o2) error of two places to two, so this place will be calculated.

Calculate

Calculate

Similarly, calculate:

Therefore:

Then, calculate

Calculate

Putting it all together:

In order to simplify the formula, sigma (h1) is used to represent the error of the hidden layer unit h1:

We can now update w1:

Repeating this for w2, w3, and w4:

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

$$w_6^+ = 0.408666186$$

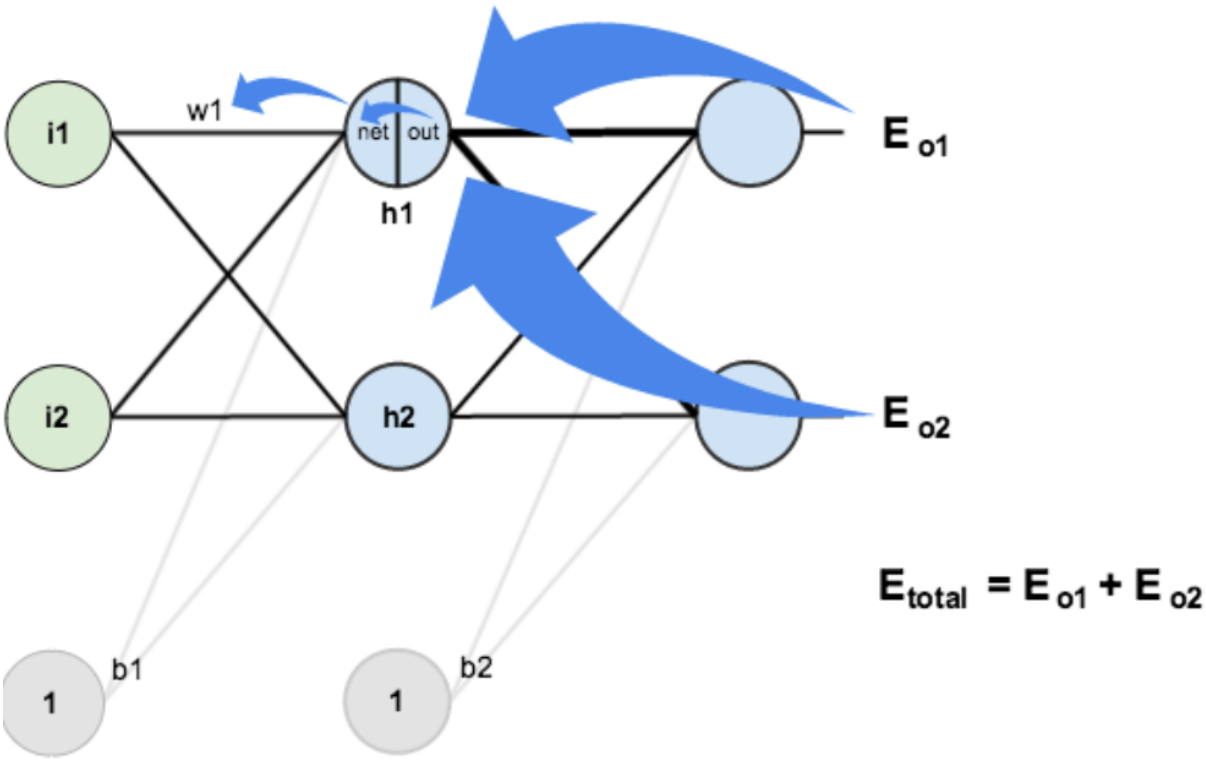
$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$$\frac{\partial E_{total}}{\partial out_{h1}}$$

$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$

$$\frac{\partial E_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$

$$\frac{\partial out_{h1}}{\partial net_{h1}}$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

$$\frac{\partial net_{h1}}{\partial w_1}$$

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \delta_o * w_{ho}) * out_{h1}(1 - out_{h1}) * i_1$$

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of back propagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.000035085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Convolution Neural Networks (Part 1)

Inputs and Outputs

When a computer sees an image (takes an image as input), it will see an array of pixel values. Depending on the resolution and size of the image, it will see a 32 x 32 x 3 array of numbers (The 3 refers to RGB values). Just to drive home the point, let's say we have a color image in JPG form and its size is 480 x 480. The representative array will be 480 x 480 x 3. Each of these numbers is given a value from 0 to 255 which describes the pixel intensity at that point. These numbers, while meaningless to us when we perform image classification, are the only inputs available to the computer. The idea is that you give the computer this array of numbers and it will output numbers that describe the probability of the image being a certain class (.80 for cat, .15 for dog, .05 for bird, etc).

What We Want the Computer to Do

When we look at a picture of a dog, we can classify it as such if the picture has identifiable features such as paws or 4 legs. In a similar way, the computer is able perform image classification by looking for low level features such as edges and curves, and then building up to more abstract concepts through a series of convolutional layers. This is a general overview of what a CNN does. Let's get into the specifics.

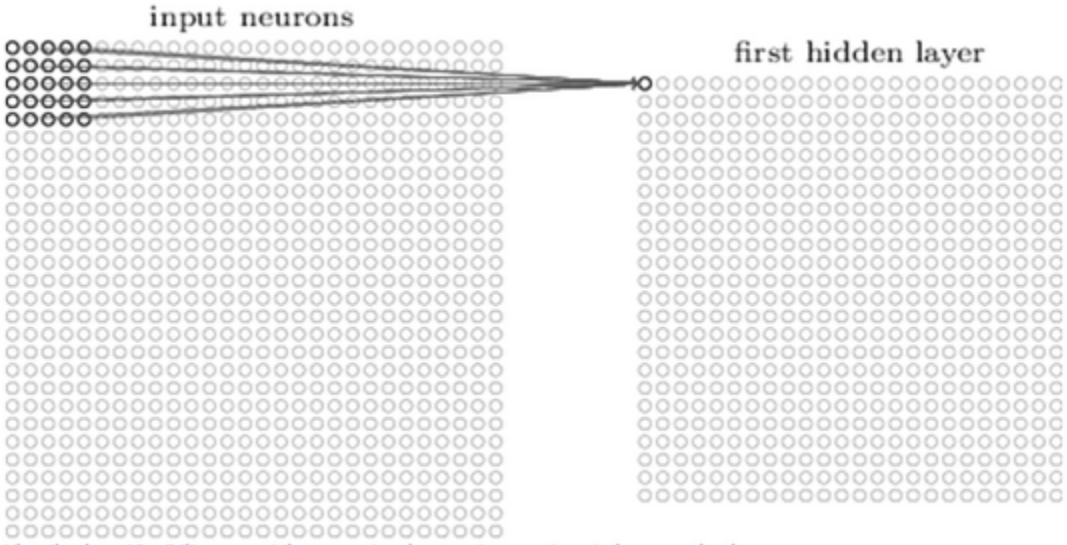
Structure

A more detailed overview of what CNNs do would be that you take the image, pass it through a series of convolutional, nonlinear, pooling (downsampling), and fully connected layers, and get an output. As we said earlier, the output can be a single class or a probability of classes that best describes the image. Now, the hard part is understanding what each of these layers do. So let's get into the most important one.

First Layer – Math Part

The first layer in a CNN is always a Convolutional Layer. First thing to make sure you remember is what the input to this conv (I'll be using that abbreviation a lot) layer is. Like we mentioned before, the input is a 32 x 32 x 3 array of pixel values. Now, the best way to explain a conv layer is to imagine a flashlight that is shining over the top left

of the image. Let's say that the light this flashlight shines covers a 5 x 5 area. And now, let's imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a filter(or sometimes referred to as a neuron or a kernel) and the region that it is shining over is called the receptive field. Now this filter is also an array of numbers (the numbers are called weights or parameters). A very important note is that the depth of this filter has to be the same as the depth of the input (this makes sure that the math works out), so the dimensions of this filter is 5 x 5 x 3. Now, let's take the first position the filter is in for example. It would be the top left corner.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

As the filter is sliding, or convolving, around the input image, it is multiplying the values in the filter with the original pixel values of the image (aka computing element wise multiplications). These multiplications are all summed up (mathematically speaking, this would be 75 multiplications in total). So now you have a single number. Remember, this number is just representative of when the filter is at the top left of the image. Now, we repeat this process for every location on the input volume. (Next step would be moving the filter to the right by 1 unit, then right again by 1, and so on). Every unique location on the input volume produces a number. After sliding the filter over all the locations, you will find out that what you're left with is a 28 x 28 x 1 array of numbers, which we call an activation map or feature map. The reason you get a 28 x 28 array is that there are 784 different locations that a 5 x 5 filter can fit on a 32 x 32 input image. These 784 numbers are mapped to a 28 x 28 array.

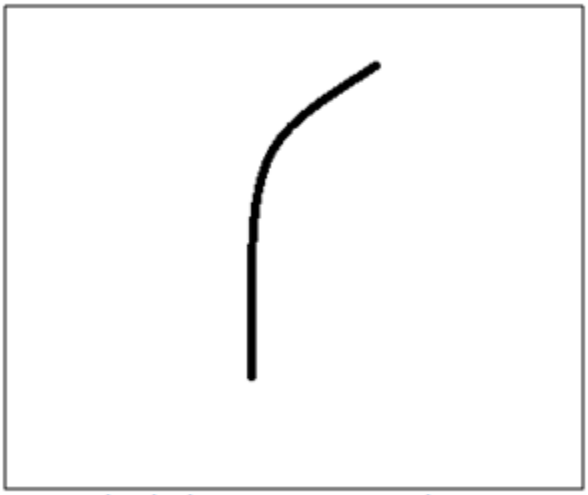
Let's say now we use two 5 x 5 x 3 filters instead of one. Then our output volume would be 28 x 28 x 2. By using more filters, we are able to preserve the spatial dimensions better. Mathematically, this is what's going on in a convolutional layer.

First Layer – High Level Perspective

However, let's talk about what this convolution is actually doing from a high level. Each of these filters can be thought of as feature identifiers. When I say features, I'm talking about things like straight edges, simple colors, and curves. Think about the simplest characteristics that all images have in common with each other. Let's say our first filter is 7 x 7 x 3 and is going to be a curve detector. (In this section, let's ignore the fact that the filter is 3 units deep and only consider the top depth slice of the filter and the image, for simplicity.)As a curve detector, the filter will have a pixel structure in which there will be higher numerical values along the area that is a shape of a curve (Remember, these filters that we're talking about as just numbers!).

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

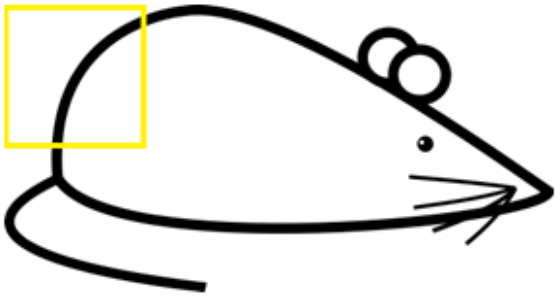
Pixel representation of filter



Visualization of a curve detector filter

Now, let's go back to visualizing this mathematically. When we have this filter at the top left corner of the input volume, it is computing multiplications between the filter and pixel values at that region. Now let's take an example of an image that we want to classify, and let's put our filter at the top left corner.

Original image



Visualization of the filter on the image

Remember, what we have to do is multiply the values in the filter with the original pixel values of the image.

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then

all of the multiplications summed together will result in a large value! Now let's see what happens when we move our filter.

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

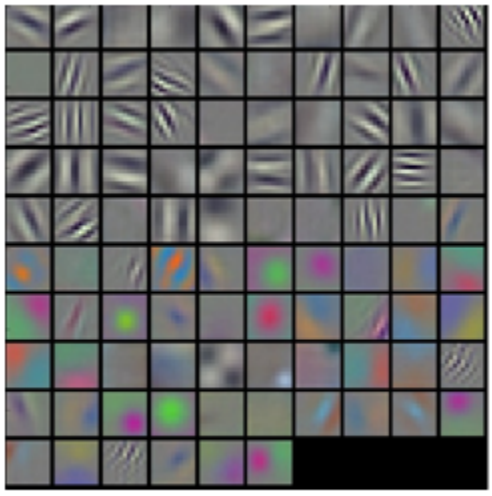
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

The value is much lower! This is because there wasn't anything in the image section that responded to the curve detector filter. Remember, the output of this conv layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at mostly likely to be curves in the picture. In this example, the top left value of our 28 x 28 x 1 activation map will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there wasn't anything in the input volume that caused the filter to activate (or more simply said, there wasn't a curve in that region of the original image). Remember, this is just for one filter. This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

Disclaimer: The filter I described in this section was simplistic for the main purpose of describing the math that goes on during a convolution. In the picture below, you'll see some examples of actual visualizations of the filters of the first conv layer of a trained network. Nonetheless, the main argument remains the same. The filters on the first layer convolve around the input image and "activate" (or compute high values) when the specific feature it is looking for is in the input volume.

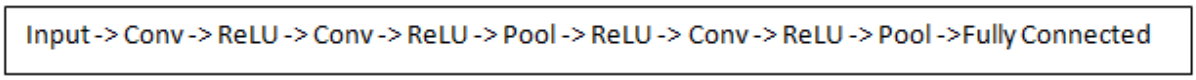


Visualizations of filters

(Quick Note: The above image came from Stanford's CS 231N course taught by Andrej Karpathy and Justin Johnson. Recommend for anyone looking for a deeper understanding of CNNs.)

Going Deeper Through the Network

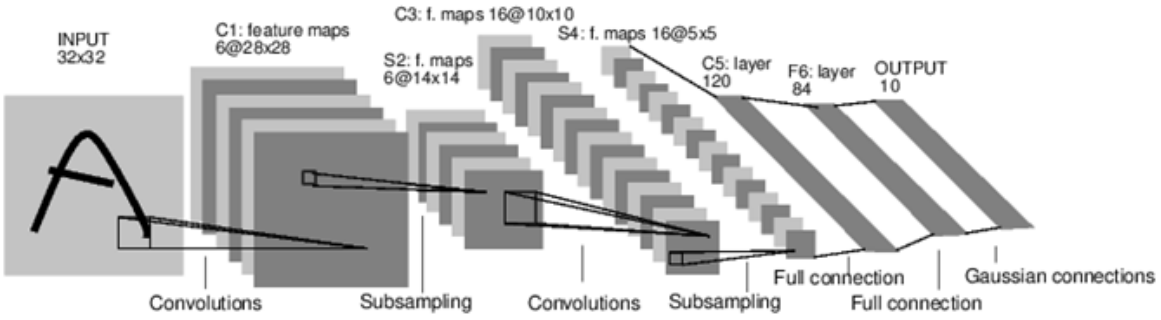
Now in a traditional convolutional neural network architecture, there are other layers that are interspersed between these conv layers. I'd strongly encourage those interested to read up on them and understand their function and effects, but in a general sense, they provide nonlinearities and preservation of dimension that help to improve the robustness of the network and control overfitting. A classic CNN architecture would look like this.



The last layer, however, is an important one and one that we will go into later on. Let's just take a step back and review what we've learned so far. We talked about what the filters in the first conv layer are designed to detect. They detect low level features such as edges and curves. As one would imagine, in order to predict whether an image is a type of object, we need the network to be able to recognize higher level features such as hands or paws or ears. So let's think about what the output of the network is after the first conv layer. It would be a 28 x 28 x 3 volume (assuming we use three 5 x 5 x 3 filters). When we go through another conv layer, the output of the first conv layer becomes the input of the 2nd conv layer. Now, this is a little bit harder to visualize. When we were talking about the first layer, the input was just the original image. However, when we're talking about the 2nd conv layer, the input is the activation map(s) that result from the first layer. So each layer of the input is basically describing the locations in the original image for where certain low level features appear. Now when you apply a set of filters on top of that (pass it through the 2nd conv layer), the output will be activations that represent higher level features. Types of these features could be semicircles (combination of a curve and straight edge) or squares (combination of several straight edges). As you go through the network and go through more conv layers, you get activation maps that represent more and more complex features. By the end of the network, you may have some filters that activate when there is handwriting in the image, filters that activate when they see pink objects, etc. If you want more information about visualizing filters in ConvNets, Matt Zeiler and Rob Fergus had an excellent research paper discussing the topic. Jason Yosinski also has a video on YouTube that provides a great visual representation. Another interesting thing to note is that as you go deeper into the network, the filters begin to have a larger and larger receptive field, which means that they are able to consider information from a larger area of the original input volume (another way of putting it is that they are more responsive to a larger region of pixel space).

Fully Connected Layer

Now that we can detect these high level features, the icing on the cake is attaching a fully connected layer to the end of the network. This layer basically takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program has to choose from. For example, if you wanted a digit classification program, N would be 10 since there are 10 digits. Each number in this N dimensional vector represents the probability of a certain class. For example, if the resulting vector for a digit classification program is [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a 10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9 (Side note: There are other ways that you can represent the output, but I am just showing the softmax approach). The way this fully connected layer works is that it looks at the output of the previous layer (which as we remember should represent the activation maps of high level features) and determines which features most correlate to a particular class. For example, if the program is predicting that some image is a dog, it will have high values in the activation maps that represent high level features like a paw or 4 legs, etc. Similarly, if the program is predicting that some image is a bird, it will have high values in the activation maps that represent high level features like wings or a beak, etc. Basically, a FC layer looks at what high level features most strongly correlate to a particular class and has particular weights so that when you compute the products between the weights and the previous layer, you get the correct probabilities for the different classes.



A Full Convolutional Neural Network (LeNet)

Training (AKA:What Makes this Stuff Work)

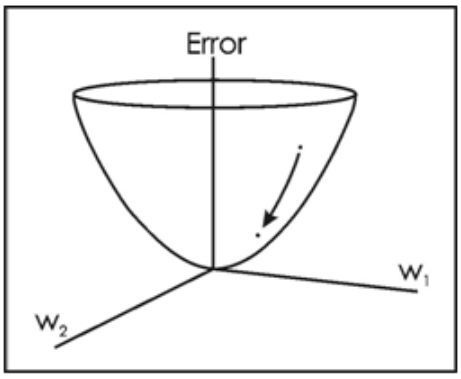
Now, this is the one aspect of neural networks that I purposely haven't mentioned yet and it is probably the most important part. There may be a lot of questions you had while reading. How do the filters in the first conv layer know to look for edges and curves? How does the fully connected layer know what activation maps to look at? How do the filters in each layer know what values to have? The way the computer is able to adjust its filter values (or weights) is through a training process called backpropagation.

Before we get into backpropagation, we must first take a step back and talk about what a neural network needs in order to work. At the moment we all were born, our minds were fresh. We didn't know what a cat or dog or bird was. In a similar sort of way, before the CNN starts, the weights or filter values are randomized. The filters don't know to look for edges and curves. The filters in the higher layers don't know to look for paws and beaks. As we grew older however, our parents and teachers showed us different pictures and images and gave us a corresponding label. This idea of being given an image and a label is the training process that CNNs go through. Before getting too into it, let's just say that we have a training set that has thousands of images of dogs, cats, and birds and each of the images has a label of what animal that picture is. Back to backprop.

So backpropagation can be separated into 4 distinct sections, the forward pass, the loss function, the backward pass, and the weight update. During the forward pass, you take a training image which as we remember is a 32 x 32 x 3 array of numbers and pass it through the whole network. On our first training example, since all of the weights or filter values were randomly initialized, the output will probably be something like [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1], basically an output that doesn't give preference to any number in particular. The network, with its current weights, isn't able to look for those low level features or thus isn't able to make any reasonable conclusion about what the classification might be. This goes to the loss function part of backpropagation. Remember that what we are using right now is training data. This data has both an image and a label. Let's say for example that the first training image inputted was a 3. The label for the image would be [0 0 0 1 0 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is ½ times (actual - predicted) squared.

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

Let's say the variable L is equal to that value. As you can imagine, the loss will be extremely high for the first couple of training images. Now, let's just think about this intuitively. We want to get to a point where the predicted label (output of the ConvNet) is the same as the training label (This means that our network got its prediction right).In order to get there, we want to minimize the amount of loss we have. Visualizing this as just an optimization problem in calculus, we want to find out which inputs (weights in our case) most directly contributed to the loss (or error) of the network.



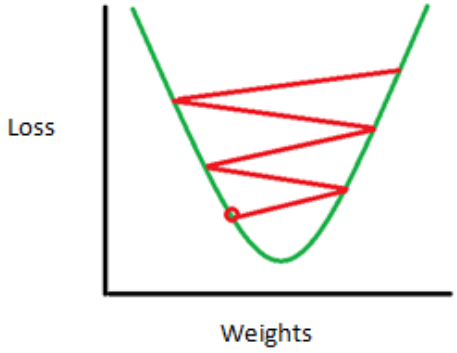
One way of visualizing this idea of minimizing the loss is to consider a 3-D graph where the weights of the neural net (there are obviously more than 2 weights, but let's go for simplicity) are the independent variables and the dependent variable is the loss. The task of minimizing the loss involves trying to adjust the weights so that the loss decreases. In visual terms, we want to get to the lowest point in our bowl shaped object. To do this, we have to take a derivative of the loss (visual terms: calculate the slope in every direction) with respect to the weights.

This is the mathematical equivalent of a dL/dW where W are the weights at a particular layer. Now, what we want to do is perform a backward pass through the network, which is determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. Once we compute this derivative, we then go to the last step which is the weight update. This is where we take all the weights of the filters and update them so that they change in the direction of the gradient.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

The learning rate is a parameter that is chosen by the programmer. A high learning rate means that bigger steps are taken in the weight updates and thus, it may take less time for the model to converge on an optimal set of weights. However, a learning rate that is too high could result in jumps that are too large and not precise enough to reach the optimal point.



Consequence of a high learning rate where the jumps are too large and we are not able to minimize the loss.

The process of forward pass, loss function, backward pass, and parameter update is generally called one epoch. The program will repeat this process for a fixed number of epochs for each set of training images (commonly called a batch). Once you finish the parameter update on the last training example, hopefully the network should be trained well enough so that the weights of the layers are tuned correctly.

Testing

Finally, to see whether or not our CNN works, we have a different set of images and labels (can't double dip between training and test!) and pass the images through the CNN. We compare the outputs to the ground truth and see if our

network works!

How Companies Use CNNs

Data, data, data. The companies that have lots of this magic 4 letter word are the ones that have an inherent advantage over the rest of the competition. The more training data that you can give to a network, the more training iterations you can make, the more weight updates you can make, and the better tuned to the network is when it goes to production. Facebook (and Instagram) can use all the photos of the billion users it currently has, Pinterest can use information of the 50 billion pins that are on its site, Google can use search data, and Amazon can use data from the millions of products that are bought every day. And now you know the magic behind how they use it.

Disclaimer

While this post should be a good start to understanding CNNs, it is by no means a comprehensive overview. Things not discussed in this post include the nonlinear and pooling layers as well as hyperparameters of the network such as filter sizes, stride, and padding. Topics like network architecture, batch normalization, vanishing gradients, dropout, initialization techniques, non-convex optimization,biases, choices of loss functions, data augmentation,regularization methods, computational considerations, modifications of backpropagation, and more were also not discussed.

Reference:

https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

Convolution Neural Networks (Part 2)

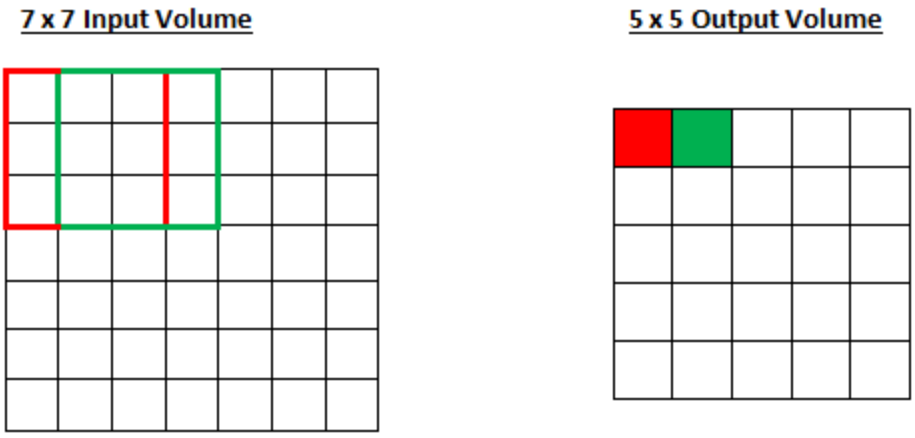
Introduction

In this post, we'll go into a lot more of the specifics of ConvNets. Disclaimer: Now, I do realize that some of these topics are quite complex and could be made in whole posts by themselves. In an effort to remain concise yet retain comprehensiveness, I will provide links to research papers where the topic is explained in more detail.

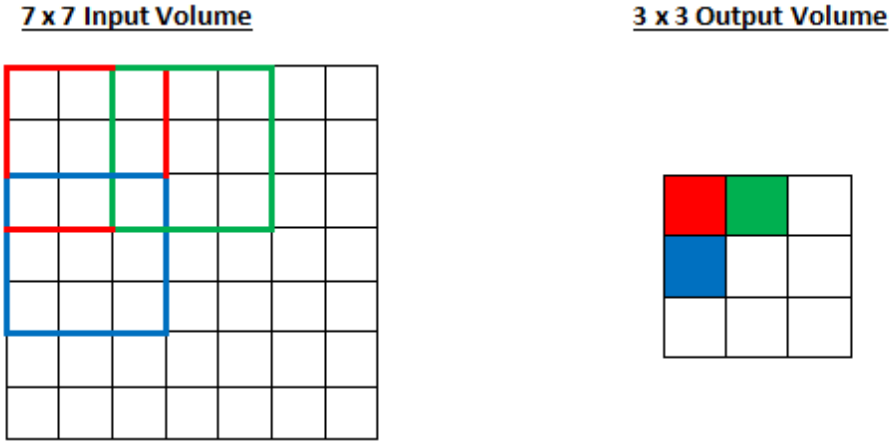
Stride and Padding

Alright, let's look back at our good old conv layers. Remember the filters, the receptive fields, the convolving? Good. Now, there are 2 main parameters that we can change to modify the behavior of each layer. After we choose the filter size, we also have to choose the stride and the padding.

Stride controls how the filter convolves around the input volume. In the example we had in part 1, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. Let's look at an example. Let's imagine a 7 x 7 input volume, a 3 x 3 filter (Disregard the 3rd dimension for simplicity), and a stride of 1. This is the case that we're accustomed to.

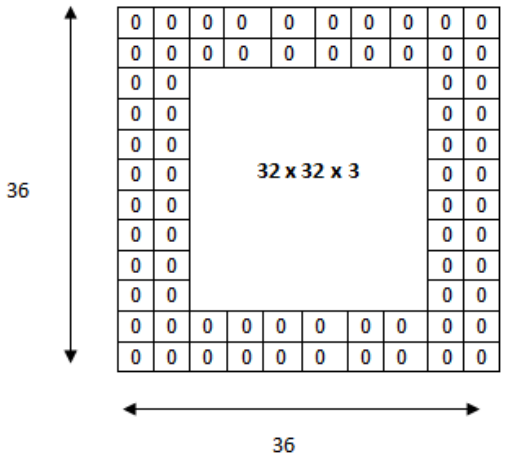


Same old, same old, right? See if you can try to guess what will happen to the output volume as the stride increases to 2.



So, as you can see, the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if we tried to set our stride to 3, then we'd have issues with spacing and making sure the receptive fields fit on the input volume. Normally, programmers will increase the stride if they want receptive fields to overlap less and if they want smaller spatial dimensions.

Now, let's take a look at padding. Before getting into that, let's think about a scenario. What happens when you apply three 5 x 5 x 3 filters to a 32 x 32 x 3 input volume? The output volume would be 28 x 28 x 3. Notice that the spatial dimensions decrease. As we keep applying conv layers, the size of the volume will decrease faster than we would like. In the early layers of our network, we want to preserve as much information about the original input volume so that we can extract those low level features. Let's say we want to apply the same conv layer but we want the output volume to remain 32 x 32 x 3. To do this, we can apply a zero padding of size 2 to that layer. Zero padding pads the input volume with zeros around the border. If we think about a zero padding of two, then this would result in a 36 x 36 x 3 input volume.



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

If you have a stride of 1 and if you set the size of zero padding to

$$Zero\ Padding = \frac{(K-1)}{2}$$

where K is the filter size, then the input and output volume will always have the same spatial dimensions.

The formula for calculating the output size for any given conv layer is

$$O = \frac{(W-K+2P)}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

Choosing Hyper-parameters

How do we know how many layers to use, how many conv layers, what are the filter sizes, or the values for stride and padding? These are not trivial questions and there isn't a set standard that is used by all researchers. This is because the network will largely depend on the type of data that you have. Data can vary by size, complexity of the image, type of image processing task, and more. When looking at your dataset, one way to think about how to choose the hyperparameters is to find the right combination that creates abstractions of the image at a proper scale.

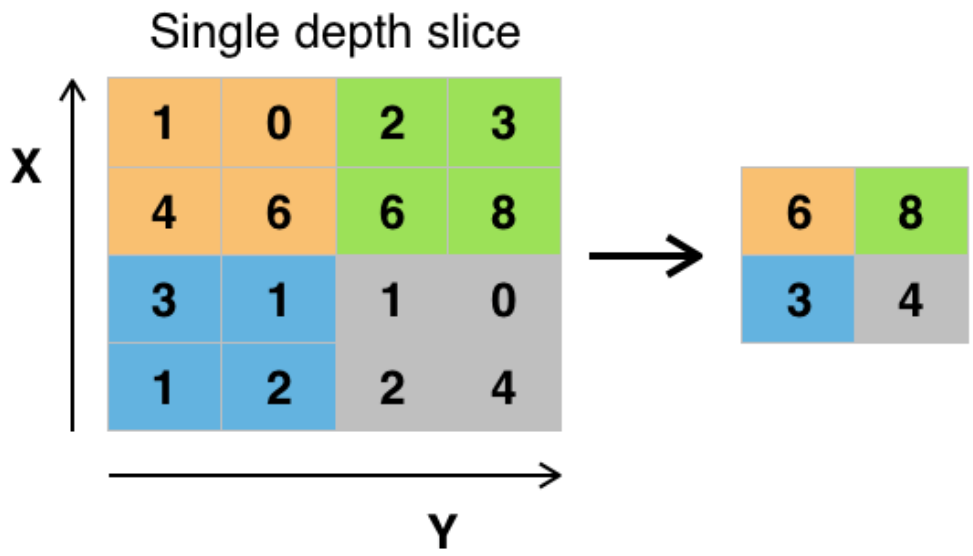
ReLU (Rectified Linear Units) Layers

After each conv layer, it is convention to apply a nonlinear layer (or activation layer) immediately afterward.The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers (Explaining this might be out of the scope of this post, but see here and here for good descriptions). The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.

Paper by the great Geoffrey Hinton (aka the father of deep learning).

Pooling Layers

After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a downsampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every subregion that the filter convolves around.



Example of Maxpool with a 2x2 filter and a stride of 2

Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control overfitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

Dropout Layers

Now, dropout layers have a very specific function in neural networks. In the last section, we discussed the problem of overfitting, where after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples. The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero in the forward pass. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that I mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

Paper by Geoffrey Hinton.

Network in Network Layers

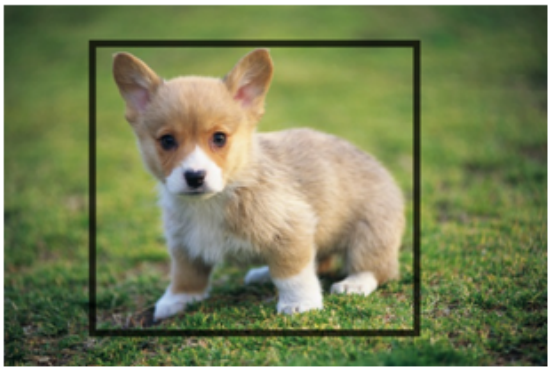
A network in network layer refers to a conv layer where a 1 x 1 size filter is used. Now, at first look, you might wonder why this type of layer would even be helpful since receptive fields are normally larger than the space they map to. However, we must remember that these 1x1 convolutions span a certain depth, so we can think of it as a 1 x 1 x N convolution where N is the number of filters applied in the layer. Effectively, this layer is performing a N-D element-wise multiplication where N is the depth of the input volume into the layer.

Paper by Min Lin.

Classification, Localization, Detection, Segmentation

In the example we used in Part 1 of this series, we looked at the task of image classification. This is the process of taking an input image and outputting a class number out of a set of categories. However, when we take a task like object localization, our job is not only to produce a class label but also a bounding box that describes where the object is in the picture.

Object Classification is the task of identifying that picture is a dog



Object Localization involves the class label as well as a bounding box to show where the object is located.

We also have the task of object detection, where localization needs to be done on all of the objects in the image. Therefore, you will have multiple bounding boxes and multiple class labels.

Finally, we also have object segmentation where the task is to output a class label as well as an outline of every object in the input image.

Reference:

https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are popular models that have shown great promise in many NLP tasks. But despite their recent popularity I've only found a limited number of resources that throughly explain how RNNs work, and how to implement them. That's what this tutorial is about.

Part 1 Introduction to RNNs

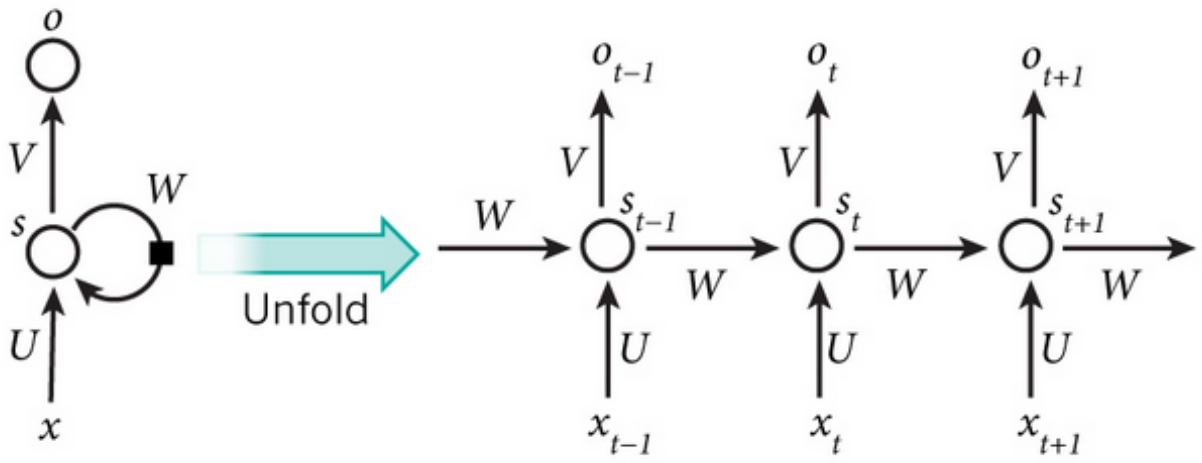
As part of the tutorial we will implement a recurrent neural network based language model. The applications of language models are two-fold: First, it allows us to score arbitrary sentences based on how likely they are to occur

in the real world. This gives us a measure of grammatical and semantic correctness. Such models are typically used as part of Machine Translation systems. Secondly, a language model allows us to generate new text (I think that's the much cooler application). Training a language model on Shakespeare allows us to generate Shakespeare-like text. This fun post by Andrej Karpathy demonstrates what character-level language models based on RNNs are capable of.

I'm assuming that you are somewhat familiar with basic Neural Networks. If you're not, you may want to head over to Implementing A Neural Network From Scratch, which guides you through the ideas and implementation behind non-recurrent networks.

WHAT ARE RNNS?

The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps (more on this later). Here is what a typical RNN looks like:



A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature

The above diagram shows a RNN being unrolled (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in a RNN are as follows:

- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step: s_t=f(Ux_t + Ws_{t-1}). The function f usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.

-o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. o_t = mathrm{softmax}(Vs_t).

There are a few things to note here:

- You can think of the hidden state s_t as the memory of the network. s_t captures information about what happened in all the previous time steps. The output at step o_t is calculated solely based on the memory at time t. As briefly mentioned above, it's a bit more complicated in practice because s_t typically can't capture information from too many time steps ago.
- Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN shares the same parameters (U, V, W above) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.
- The above diagram has outputs at each time step, but depending on the task this may not be necessary. For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word. Similarly, we may not need inputs at each time step. The main feature of an RNN is its hidden state, which captures some information about a sequence.

What can RNNs do?

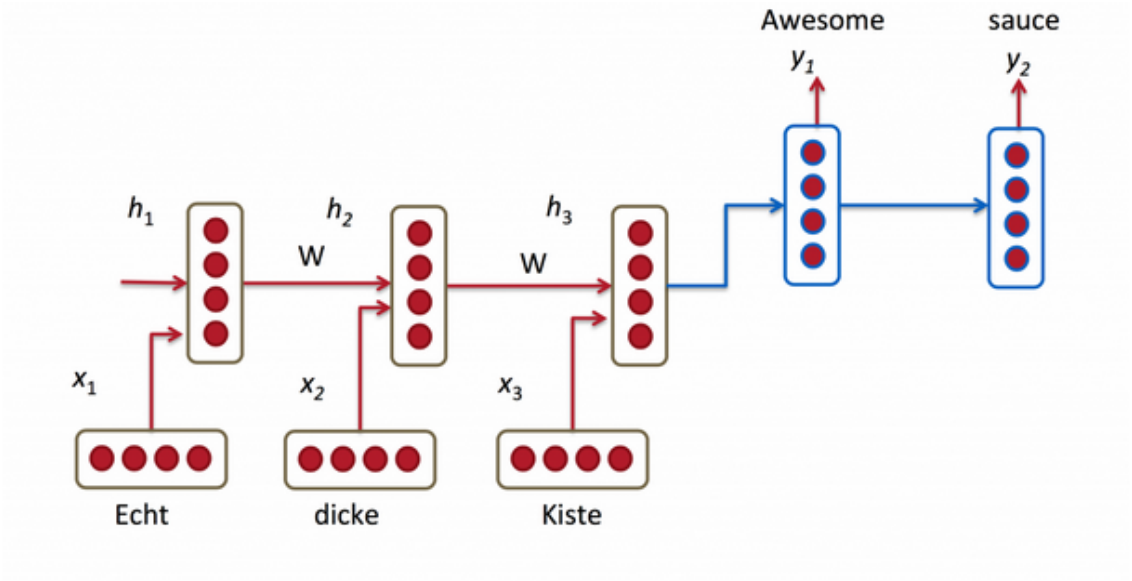
RNNs have shown great success in many NLP tasks. At this point I should mention that the most commonly used type of RNNs are LSTMs, which are much better at capturing long-term dependencies than vanilla RNNs are. But don't worry, LSTMs are essentially the same thing as the RNN we will develop in this tutorial, they just have a different way of computing the hidden state. We'll cover LSTMs in more detail in a later post. Here are some example applications of RNNs in NLP (by non means an exhaustive list).

Language Modeling and Generating Text

Given a sequence of words we want to predict the probability of each word given the previous words. Language Models allow us to measure how likely a sentence is, which is an important input for Machine Translation (since high-probability sentences are typically correct). A side-effect of being able to predict the next word is that we get a generative model, which allows us to generate new text by sampling from the output probabilities. And depending on what our training data is we can generate all kinds of stuff. In Language Modeling our input is typically a sequence of words (encoded as one-hot vectors for example), and our output is the sequence of predicted words. When training the network we set o_t = x_{t+1} since we want the output at step t to be the actual next word.

Machine Translation

Machine Translation is similar to language modeling in that our input is a sequence of words in our source language (e.g. German). We want to output a sequence of words in our target language (e.g. English). A key difference is that our output only starts after we have seen the complete input, because the first word of our translated sentences may require information captured from the complete input sequence.



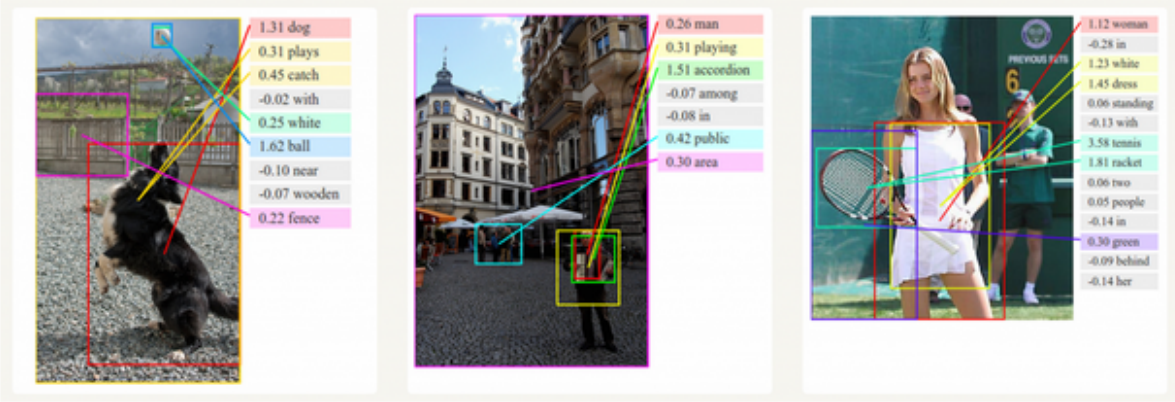
RNN for Machine Translation. Image Source: http://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf

Speech Recognition

Given an input sequence of acoustic signals from a sound wave, we can predict a sequence of phonetic segments together with their probabilities.

Generating Image Descriptions

Together with convolutional Neural Networks, RNNs have been used as part of a model to generate descriptions for unlabeled pics. It's quite amazing how well this seems to work. The combined model even aligns the generated words with features found in the pics.



Deep Visual-Semantic Alignments for Generating Image Descriptions. Source: http://cs.stanford.edu/people/karpathy/deepimagesent/

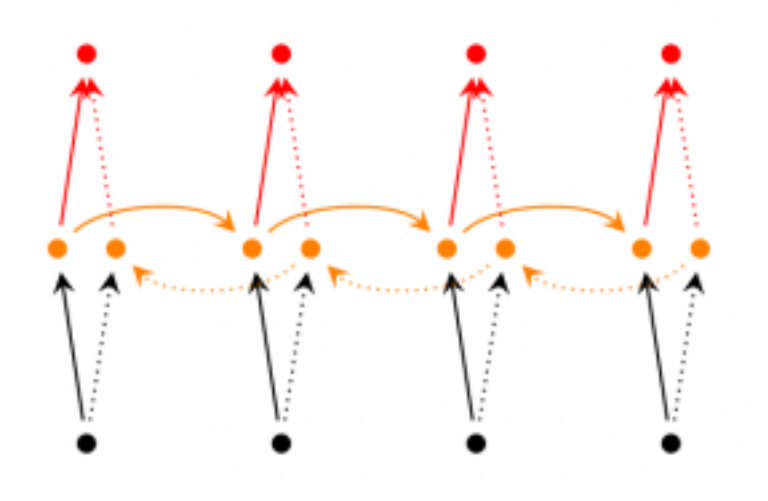
Training RNNs

Training a RNN is similar to training a traditional Neural Network. We also use the backpropagation algorithm, but with a little twist. Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. For example, in order to calculate the gradient at t=4 we would need to backpropagate 3 steps and sum up the gradients. This is called Backpropagation Through Time (BPTT). If this doesn't make a whole lot of sense yet, don't worry, we'll have a whole post on the gory details. For now, just be aware of the fact that vanilla RNNs trained with BPTT have difficulties learning long-term dependencies (e.g. dependencies between steps that are far apart) due to what is called the vanishing/exploding gradient problem. There exists some machinery to deal with these problems, and certain types of RNNs (like LSTMs) were specifically designed to get around them.

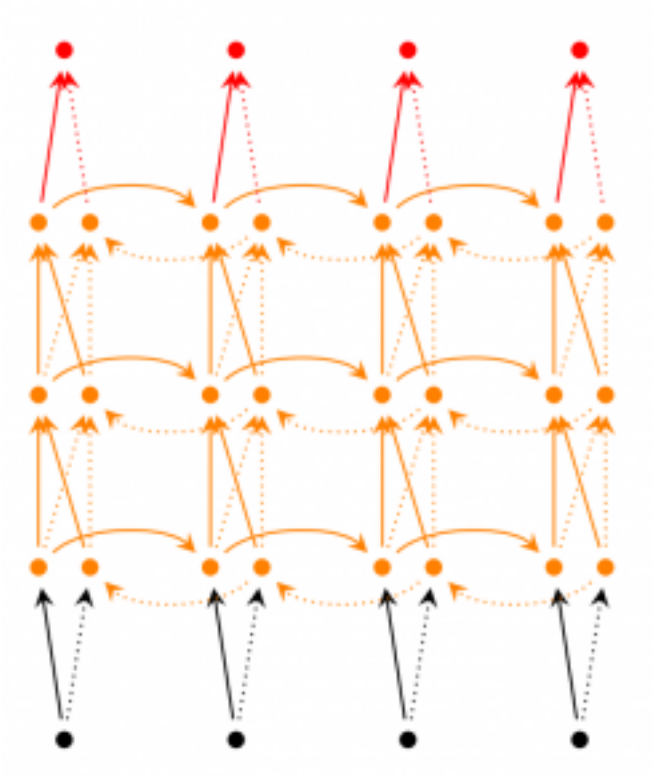
RNN Extensions

Over the years researchers have developed more sophisticated types of RNNs to deal with some of the shortcomings of the vanilla RNN model. We will cover them in more detail in a later post, but I want this section to serve as a brief overview so that you are familiar with the taxonomy of models.

Bidirectional RNNs are based on the idea that the output at time t may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs.



Deep (Bidirectional) RNNs are similar to Bidirectional RNNs, only that we now have multiple layers per time step. In practice this gives us a higher learning capacity (but we also need a lot of training data).



LSTM networks are quite popular these days and we briefly talked about them above. LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. The memory in LSTMs are called cells and you can think of them as black boxes that take as input the previous state h_{t-1} and current input x_t. Internally these cells decide what to keep in (and what to erase from) memory. They then combine the previous state, the current memory, and the input. It turns out that these types of units are very efficient at capturing long-term dependencies. LSTMs can be quite confusing in the beginning but if you're interested in learning more this post has an excellent explanation.

Conclusion

So far so good. I hope you've gotten a basic understanding of what RNNs are and what they can do. In the next post we'll implement a first version of our language model RNN using Python and Theano. Please leave questions in the comments!

Part 2 Implementing a RNN with Python, Numpy and Theano

In this part we will implement a full Recurrent Neural Network from scratch using Python and optimize our implementation using Theano, a library to perform operations on a GPU. The full code is available on Github. I will skip over some boilerplate code that is not essential to understanding Recurrent Neural Networks, but all of that is also on Github.

Language Modeling

Our goal is to build a Language Model using a Recurrent Neural Network. Here's what that means. Let's say we have sentence of m words. A language model allows us to predict the probability of observing the sentence (in a given dataset) as:

begin{aligned} P(w_1,...,w_m) = prod_{i=1}^{m} P(w_i mid w_1,..., w_{i-1}) end{aligned}

In words, the probability of a sentence is the product of probabilities of each word given the words that came before it. So, the probability of the sentence "He went to buy some chocolate" would be the probability of "chocolate" given "He went to buy some", multiplied by the probability of "some" given "He went to buy", and so on.

Why is that useful? Why would we want to assign a probability to observing a sentence?

First, such a model can be used as a scoring mechanism. For example, a Machine Translation system typically generates multiple candidates for an input sentence. You could use a language model to pick the most probable sentence. Intuitively, the most probable sentence is likely to be grammatically correct. Similar scoring happens in speech recognition systems.

But solving the Language Modeling problem also has a cool side effect. Because we can predict the probability of a word given the preceding words, we are able to generate new text. It's a generative model. Given an existing sequence of words we sample a next word from the predicted probabilities, and repeat the process until we have a full sentence. Andrej Karparthy has a great post that demonstrates what language models are capable of. His models are trained on single characters as opposed to full words, and can generate anything from Shakespeare to Linux Code.

Note that in the above equation the probability of each word is conditioned on all previous words. In practice, many models have a hard time representing such long-term dependencies due to computational or memory constraints. They are typically limited to looking at only a few of the previous words. RNNs can, in theory, capture such long-term dependencies, but in practice it's a bit more complex. We'll explore that in a later post.

Training Data and Preprocessing

To train our language model we need text to learn from. Fortunately we don't need any labels to train a language model, just raw text. I downloaded 15,000 longish reddit comments from a dataset available on Google's BigQuery. Text generated by our model will sound like reddit commenters (hopefully)! But as with most Machine Learning projects we first need to do some pre-processing to get our data into the right format.

1. Tokenize Text

We have raw text, but we want to make predictions on a per-word basis. This means we must tokenize our comments into sentences, and sentences into words. We could just split each of the comments by spaces, but that wouldn't handle punctuation properly. The sentence "He left!" should be 3 tokens: "He", "left", "!". We'll use NLTK's word_tokenize and sent_tokenize methods, which do most of the hard work for us.

2. Remove infrequent words

Most words in our text will only appear one or two times. It's a good idea to remove these infrequent words. Having a huge vocabulary will make our model slow to train (we'll talk about why that is later), and because we don't have a lot of contextual examples for such words we wouldn't be able to learn how to use them correctly anyway. That's quite similar to how humans learn. To really understand how to appropriately use a word you need to have seen it in different contexts.

In our code we limit our vocabulary to the vocabulary_size most common words (which I set to 8000, but feel free to change it). We replace all words not included in our vocabulary by UNKNOWN_TOKEN. For example, if we don't include the word "nonlinearities" in our vocabulary, the sentence "nonlinerities are important in neural networks" becomes "UNKNOWN_TOKEN are important in Neural Networks". The word UNKNOWN_TOKEN will become part of our vocabulary and we will predict it just like any other word. When we generate new text we can replace UNKNOWN_TOKEN again, for example by taking a randomly sampled word not in our vocabulary, or we could just generate sentences until we get one that doesn't contain an unknown token.

3. Prepend special start and end tokens

We also want to learn which words tend start and end a sentence. To do this we prepend a special SENTENCE_START token, and append a special SENTENCE_END token to each sentence. This allows us to ask: Given that the first token is SENTENCE_START, what is the likely next word (the actual first word of the sentence)?

4. Build training data matrices

The input to our Recurrent Neural Networks are vectors, not strings. So we create a mapping between words and indices, index_to_word, and word_to_index. For example, the word "friendly" may be at index 2001. A training example x may look like [0, 179, 341, 416], where 0 corresponds to SENTENCE_START. The corresponding label y would be [179, 341, 416, 1]. Remember that our goal is to predict the next word, so y is just the x vector shifted by one position with the last element being the SENTENCE_END token. In other words, the correct prediction for word 179 above would be 341, the actual next word.

```
import os
import csv
import itertools

import nltk
import numpy as np

import npdl


def load_data(corpus_path=os.path.join(os.path.dirname(__file__), 'data/lm/reddit-
↪comments-2015-08.csv'),
              vocabulary_size=8000):
    sentence_start_token = "SENTENCE_START"
    sentence_end_token = "SENTENCE_END"
    unknown_token = 'UNKNOWN_TOKEN'

    # Read the data and append SENTENCE_START and SENTENCE_END tokens
    with open(corpus_path, encoding='utf-8') as f:
        reader = csv.reader(f, skipinitialspace=True)
        # Split full comments into sentences
        sentences = [nltk.sent_tokenize(x[0]) for x in reader]
        sentences = itertools.chain(*sentences)
        # Append SENTENCE_START and SENTENCE_END
        sentences = ["%s %s %s" % (sentence_start_token, x, sentence_end_token) for x␣
↪in sentences]

    # Tokenize the sentences into words
    tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in sentences]

    # Count the word frequencies
    word_freq = nltk.FreqDist(itertools.chain(*tokenized_sentences))
    print("Found %d unique tokens in corpus '%s'." % (word_freq.B(), corpus_path))

    # Get the most common words and build index_to_word and word_to_index vectors
    vocab = word_freq.most_common(vocabulary_size - 1)
    print("The least frequent word in our vocabulary is '%s' and appeared %d times." %
          (vocab[-1][0], vocab[-1][1]))
    index_to_word = [x[0] for x in vocab]
    index_to_word.append(unknown_token)
    word_to_index = dict([(w, i) for i, w in enumerate(index_to_word)])
```

```
    # Replace all words not in our vocabulary with the unknown token
    tokenized_sentences = [[word if word in word_to_index else unknown_token for word␣
↪in sentence]
                           for sentence in tokenized_sentences]
    # Create the training data
    train_x = np.asarray([[word_to_index[word] for word in sentence[:-1]] for␣
↪sentence in tokenized_sentences])
    train_y = np.asarray([[word_to_index[word] for word in sentence[1:]] for sentence␣
↪in tokenized_sentences])

    return index_to_word, word_to_index, train_x, train_y


def main(max_iter, corpus_path=os.path.join(os.path.dirname(__file__), 'data/lm/tiny_
↪shakespeare.txt')):
    raw_text = open(corpus_path, 'r').read()
    chars = list(set(raw_text))
    data_size, vocab_size = len(raw_text), len(chars)
    print("data has %s charactres, % unique." % (data_size, vocab_size))
    char_to_index = {ch: i for i, ch in enumerate(chars)}
    index_to_char = {i: ch for i, ch in enumerate(chars)}

    time_steps, batch_size = 30, 40

    length = batch_size * 20
    text_pointers = np.random.randint(data_size - time_steps - 1, size=length)
    batch_in = np.zeros([length, time_steps, vocab_size])
    batch_out = np.zeros([length, vocab_size], dtype=np.uint8)
    for i in range(length):
        b_ = [char_to_index[c] for c in raw_text[text_pointers[i]:text_pointers[i] +␣
↪time_steps + 1]]
        batch_in[i, range(time_steps), b_[:-1]] = 1
        batch_out[i, b_[-1]] = 1

    print("Building model ...")
    net = npdl.Model()
    # net.add(model.layers.SimpleRNN(n_out=500, return_sequence=True))
    net.add(npdl.layers.SimpleRNN(n_out=500, n_in=vocab_size))
    net.add(npdl.layers.Softmax(n_out=vocab_size))
    net.compile(loss=npdl.objectives.SCCE(), optimizer=npdl.optimizers.SGD(lr=0.00001,
↪ clip=5))

    print("Train model ...")
    net.fit(batch_in, batch_out, max_iter=max_iter, batch_size=batch_size)


if __name__ == '__main__':
    main(100)
```

CHAPTER 3

Examples

This part provides examples for building deep neural networks.

CHAPTER 4

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

npdl.activation

Non-linear activation functions for artificial neurons.

A function used to transform the activation level of a unit (neuron) into an output signal. Typically, activation functions have a "squashing" effect. Together with the PSP function (which is applied first) this defines the unit type. Neural Networks supports a wide range of activation functions.

Activations

Activation()	Base class for activations.
Sigmoid()	Sigmoid activation function.
Tanh()	Tanh activation function.
ReLU()	Rectify activation function.
Linear()	Linear activation function.
Softmax()	Softmax activation function.
Elliot([steepness])	A fast approximation of sigmoid.
SymmetricElliot([steepness])	Elliot symmetric sigmoid transfer function.
SoftPlus()	Softplus activation function.
SoftSign()	SoftSign activation function.

Detailed Description

class npdl.activation.**Activation**
Base class for activations.

derivative()

Backward step.

forward(*input*)
Forward Step.

Parameters **input** : numpy.array

the input matrix.

class npdl.activation.**Sigmoid**
Sigmoid activation function.

derivative()
The derivative of sigmoid is

$$\begin{aligned}\frac{dy}{dx} &= (1 - \varphi(x)) \otimes \varphi(x) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{e^{x}}{(1 + e^{x})^2}\end{aligned}$$

Returns float32

The derivative of sigmoid function.

forward(*input*, **args*, ***kwargs*)
A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. Often, sigmoid function refers to the special case of the logistic function and defined by the formula $\varphi(x) = \frac{1}{1+e^{-x}}$ (given the input x).

Parameters **input** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [0, 1]

The output of the sigmoid function applied to the activation.

```
class Sigmoid(Activation):
    """Sigmoid activation function.
    """

    def __init__(self):
        super(Sigmoid, self).__init__()

    def forward(self, input, *args, **kwargs):
        """A sigmoid function is a mathematical function having a
        characteristic "S"-shaped curve or sigmoid curve. Often,
        sigmoid function refers to the special case of the logistic
        function and defined by the formula :math:`\\varphi(x) = \\frac{1}{1 + e^{-x}}
↪`
        (given the input :math:`x`).

        Parameters
        ----------
        input : float32
            The activation (the summed, weighted input of a neuron).

        Returns
```

```
        -------
        float32 in [0, 1]
            The output of the sigmoid function applied to the activation.
        """

        self.last_forward = 1.0 / (1.0 + np.exp(-input))
        return self.last_forward

    def derivative(self):
        """The derivative of sigmoid is

        .. math:: \\frac{dy}{dx} & = (1-\\varphi(x)) \\otimes \\varphi(x)  \\\\
                  & = \\frac{e^{-x}}{(1+e^{-x})^2} \\\\
                  & = \\frac{e^x}{(1+e^x)^2}

        Returns
        -------
        float32
            The derivative of sigmoid function.
        """
        return np.multiply(self.last_forward, 1 - self.last_forward)
```

class npdl.activation.**Tanh**

Tanh activation function.

The hyperbolic tangent function is an old mathematical function. It was first used in the work by L'Abbe Sauri (1774).

derivative()

The derivative of `tanh()` functions is

$$\begin{aligned}\frac{d}{dx}tanh(x) &= \frac{d}{dx}\frac{sinh(x)}{cosh(x)} \\ &= \frac{cosh(x)\frac{d}{dx}sinh(x) - sinh(x)\frac{d}{dx}cosh(x)}{cosh^2(x)} \\ &= \frac{cosh(x)cosh(x) - sinh(x)sinh(x)}{cosh^2(x)} \\ &= 1 - tanh^2(x)\end{aligned}$$

Returns float32

The derivative of tanh function.

forward(*input*)

This function is easily defined as the ratio between the hyperbolic sine and the cosine functions (or expanded, as the ratio of the halfdifference and halfsum of two exponential functions in the points z and $-z$):

$$\begin{aligned}tanh(z) &= \frac{sinh(z)}{cosh(z)} \\ &= \frac{e^z - e^{-z}}{e^z + e^{-z}}\end{aligned}$$

Fortunately, numpy provides `tanh()` methods. So in our implementation, we directly use $\varphi(x) = \tanh(x)$.

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 in [-1, 1]

The output of the tanh function applied to the activation.

```
class Tanh(Activation):
    """Tanh activation function.

    The hyperbolic tangent function is an old mathematical function.
    It was first used in the work by L'Abbe Sauri (1774).
    """

    def __init__(self):
        super(Tanh, self).__init__()

    def forward(self, input):
        """This function is easily defined as the ratio between the hyperbolic
        sine and the cosine functions (or expanded, as the ratio of the
        halfdifference and halfsum of two exponential functions in the
        points :math:`z` and :math:`-z`):

        .. math:: tanh(z) & = \\frac{sinh(z)}{cosh(z)} \\\\
                  & = \\frac{e^z - e^{-z}}{e^z + e^{-z}}

        Fortunately, numpy provides :meth:`tanh` methods. So in our implementation,
        we directly use :math:`\\varphi(x) = \\tanh(x)`.

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32 in [-1, 1]
            The output of the tanh function applied to the activation.
        """
        self.last_forward = np.tanh(input)
        return self.last_forward

    def derivative(self):
        """The derivative of :meth:`tanh` functions is

        .. math:: \\frac{d}{dx} tanh(x) & = \\frac{d}{dx} \\frac{sinh(x)}{cosh(x)}␣
↪\\\\
                  & = \\frac{cosh(x) \\frac{d}{dx}sinh(x) - sinh(x) \\frac{d}{dx}
↪cosh(x) }{ cosh^2(x)} \\\\
                  & = \\frac{ cosh(x) cosh(x) - sinh(x) sinh(x) }{ cosh^2(x)}  \\\\
                  & = 1 - tanh^2(x)

        Returns
        -------
        float32
            The derivative of tanh function.
        """
```

```
        return 1 - np.power(self.last_forward, 2)
```

class npdl.activation.**ReLU**

Rectify activation function.

Two additional major benefits of ReLUs are sparsity and a reduced likelihood of vanishing gradient. But first recall the definition of a ReLU is $h = max(0, a)$ where $a = Wx + b$.

One major benefit is the reduced likelihood of the gradient to vanish. This arises when $a > 0$. In this regime the gradient has a constant value. In contrast, the gradient of sigmoids becomes increasingly small as the absolute value of x increases. The constant gradient of ReLUs results in faster learning.

The other benefit of ReLUs is sparsity. Sparsity arises when $a0$. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.

derivative()

The point-wise derivative for ReLU is $\frac{dy}{dx} = 1$, if $x > 0$, or $\frac{dy}{dx} = 0$, if $x <= 0$.

Returns float32

The derivative of ReLU function.

forward(*input*)

During the forward pass, it inhibits all inhibitions below some threshold , typically 0. In other words, it computes point-wise

$$y = max(0, x)$$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the rectify function applied to the activation.

```
class ReLU(Activation):
    """Rectify activation function.

    Two additional major benefits of ReLUs are sparsity and a reduced
    likelihood of vanishing gradient. But first recall the definition
    of a ReLU is :math:`h=max(0,a)` where :math:`a=Wx+b`.

    One major benefit is the reduced likelihood of the gradient to vanish.
    This arises when :math:`a>0`. In this regime the gradient has a constant value.
    In contrast, the gradient of sigmoids becomes increasingly small as the
    absolute value of :math:`x` increases. The constant gradient of ReLUs results in
    faster learning.

    The other benefit of ReLUs is sparsity. Sparsity arises when :math:`a0`.
    The more such units that exist in a layer the more sparse the resulting
    representation. Sigmoids on the other hand are always likely to generate
    some non-zero value resulting in dense representations. Sparse representations
    seem to be more beneficial than dense representations.
    """
```

```
    def __init__(self):
        super(ReLU, self).__init__()

    def forward(self, input):
        """During the forward pass, it inhibits all inhibitions below some
        threshold :math:``, typically :math:`0`. In other words, it computes point-
↪wise

        .. math:: y=max(0,x)

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32
            The output of the rectify function applied to the activation.
        """
        self.last_forward = input
        return np.maximum(0.0, input)

    def derivative(self):
        """The point-wise derivative for ReLU is :math:`\\frac{dy}{dx} = 1`, if
        :math:`x>0`, or :math:`\\frac{dy}{dx} = 0`, if :math:`x<=0`.

        Returns
        -------
        float32
            The derivative of ReLU function.
        """
        res = np.zeros(self.last_forward.shape, dtype=get_dtype())
        res[self.last_forward > 0] = 1.
        return res
```

class npdl.activation.**Linear**

Linear activation function.

derivative()

The backward also return identity matrix.

Returns float32

The derivative of linear function.

forward(*input*)

It's also known as identity activation funtion. The forward step is $\varphi(x) = x$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the identity applied to the activation.

```
class Linear(Activation):
    """Linear activation function.
    """

    def __init__(self):
        super(Linear, self).__init__()

    def forward(self, input):
        """It's also known as identity activation funtion. The
        forward step is :math:`\\varphi(x) = x`

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32
            The output of the identity applied to the activation.
        """
        self.last_forward = input
        return input

    def derivative(self):
        """The backward also return identity matrix.

        Returns
        -------
        float32
            The derivative of linear function.
        """
        return np.ones(self.last_forward.shape, dtype=get_dtype())
```

class npdl.activation.**Softmax**

Softmax activation function.

derivative()

Returns float32

The derivative of Softmax function.

forward(*input*)

$\varphi(\mathbf{x})_j = \frac{e^{\mathbf{x}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}_k}}$ where K is the total number of neurons in the layer. This activation function gets applied row-wise.

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32 where the sum of the row is 1 and each single value is in [0, 1]

The output of the softmax function applied to the activation.

```
class Softmax(Activation):
    """Softmax activation function.
    """

    def __init__(self):
        super(Softmax, self).__init__()

    def forward(self, input):
        """:math:`\\varphi(\\mathbf{x})_j =
        \\frac{e^{\mathbf{x}_j}}{\sum_{k=1}^K e^{\mathbf{x}_k}}`
        where :math:`K` is the total number of neurons in the layer. This
        activation function gets applied row-wise.

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32 where the sum of the row is 1 and each single value is in [0, 1]
            The output of the softmax function applied to the activation.
        """
        assert np.ndim(input) == 2
        self.last_forward = input
        x = input - np.max(input, axis=1, keepdims=True)
        exp_x = np.exp(x)
        s = exp_x / np.sum(exp_x, axis=1, keepdims=True)
        return s

    def derivative(self):
        """

        Returns
        -------
        float32
            The derivative of Softmax function.
        """
        return np.ones(self.last_forward.shape, dtype=get_dtype())
```

class npdl.activation.**Elliot**(*steepness=1*)

A fast approximation of sigmoid.

The function was first introduced in 1993 by D.L. Elliot under the title A Better Activation Function for Artificial Neural Networks. The function closely approximates the Sigmoid or Hyperbolic Tangent functions for small values, however it takes longer to converge for large values (i.e. It doesn't go to 1 or 0 as fast), though this isn't particularly a problem if you're using it for classification.

derivative()

Returns float32

The derivative of Elliot function.

```
class Elliot(Activation):
    """ A fast approximation of sigmoid.

    The function was first introduced
    in 1993 by D.L. Elliot under the title A Better Activation Function for
    Artificial Neural Networks. The function closely approximates the
    Sigmoid or Hyperbolic Tangent functions for small values, however it
    takes longer to converge for large values (i.e. It doesn't go to 1 or
    0 as fast), though this isn't particularly a problem if you're using
    it for classification.

    """

    def __init__(self, steepness=1):
        super(Elliot, self).__init__()

        self.steepness = steepness

    def forward(self, input):
        self.last_forward = 1 + np.abs(input * self.steepness)
        return 0.5 * self.steepness * input / self.last_forward + 0.5

    def derivative(self):
        """

        Returns
        -------
        float32
            The derivative of Elliot function.
        """
        return 0.5 * self.steepness / np.power(self.last_forward, 2)
```

class npdl.activation.**SymmetricElliot**(*steepness=1*)

Elliot symmetric sigmoid transfer function.

derivative()

Returns float32

The derivative of SymmetricElliot function.

```
class SymmetricElliot(Activation):
    """Elliot symmetric sigmoid transfer function.

    """
    def __init__(self, steepness=1):
        super(SymmetricElliot, self).__init__()
        self.steepness = steepness

    def forward(self, input):
        self.last_forward = 1 + np.abs(input * self.steepness)
        return input * self.steepness / self.last_forward

    def derivative(self):
        """
```

```
        Returns
        -------
        float32
            The derivative of SymmetricElliot function.
        """
        return self.steepness / np.power(self.last_forward, 2)
```

class npdl.activation.**SoftPlus**

Softplus activation function.

derivative()

Returns float32

The derivative of Softplus function.

forward(*input*)

$\varphi(x) = \log(1 + e^x)$

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the softplus function applied to the activation.

```
class SoftPlus(Activation):
    """Softplus activation function.
    """

    def __init__(self):
        super(SoftPlus, self).__init__()

    def forward(self, input):
        """:math:`\\varphi(x) = \\log(1 + e^x)`

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32
            The output of the softplus function applied to the activation.
        """
        self.last_forward = np.exp(input)
        return np.log(1 + self.last_forward)

    def derivative(self):
        """

        Returns
        -------
        float32
            The derivative of Softplus function.
```

```
        """
        return self.last_forward / (1 + self.last_forward)
```

class npdl.activation.**SoftSign**

SoftSign activation function.

derivative()

Returns float32

The derivative of SoftSign function.

forward(*input*)

Parameters **x** : float32

The activation (the summed, weighted input of a neuron).

Returns float32

The output of the softplus function applied to the activation.

```
class SoftSign(Activation):
    """SoftSign activation function.

    """
    def __init__(self):
        super(SoftSign, self).__init__()

    def forward(self, input):
        """

        Parameters
        ----------
        x : float32
            The activation (the summed, weighted input of a neuron).

        Returns
        -------
        float32
            The output of the softplus function applied to the activation.
        """
        self.last_forward = np.abs(input) + 1
        return input / self.last_forward

    def derivative(self):
        """

        Returns
        -------
        float32
            The derivative of SoftSign function.
        """
        return 1. / np.power(self.last_forward, 2)
```

npdl.initialization

Functions to create initializers for parameter variables.

Examples

```
>>> from npdl.layers import Dense
>>> from npdl.initialization import GlorotUniform
>>> l1 = Dense(n_out=300, n_in=100, init=GlorotUniform())
```

Initializers

Zero	Initialize weights with zero value.
One	Initialize weights with one value.
Uniform([scale])	Sample initial weights from the uniform distribution.
Normal([std, mean])	Sample initial weights from the Gaussian distribution.
Orthogonal([gain])	Intialize weights as Orthogonal matrix.

Detailed Description

class npdl.initialization.**Initializer**

Base class for parameter weight initializers.

The *Initializer* class represents a weight initializer used to initialize weight parameters in a neural network layer. It should be subclassed when implementing new types of weight initializers.

call(*size*)

Sample should return a numpy.array of size shape and data type numpy.float32.

Parameters **size** : tuple or int.

Integer or tuple specifying the size of the returned matrix.

Returns numpy.array.

Matrix of size shape and dtype numpy.float32.

class npdl.initialization.**Zero**

Initialize weights with zero value.

class npdl.initialization.**One**

Initialize weights with one value.

class npdl.initialization.**Normal**(*std=0.01*, *mean=0.0*)

Sample initial weights from the Gaussian distribution.

Initial weight parameters are sampled from N(mean, std).

Parameters **std** : float.

Std of initial parameters.

mean : float.

Mean of initial parameters.

class npdl.initialization.**Uniform**(*scale=0.05*)
Sample initial weights from the uniform distribution.

Parameters are sampled from U(a, b).

Parameters **scale** : float or tuple.

When std is None then range determines a, b. If range is a float the weights are sampled from U(-range, range). If range is a tuple the weights are sampled from U(range[0], range[1]).

class npdl.initialization.**Orthogonal**(*gain=1.0*)
Intialize weights as Orthogonal matrix.

Orthogonal matrix initialization *[R2]*. For n-dimensional shapes where n > 2, the n-1 trailing axes are flattened. For convolutional layers, this corresponds to the fan-in, so this makes the initialization usable for both dense and convolutional layers.

Parameters **gain** : float or 'relu'.

Scaling factor for the weights. Set this to `1.0` for linear and sigmoid units, to 'relu' or `sqrt(2)` for rectified linear units, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. Other transfer functions may need different factors.

References

[R2]

class npdl.initialization.**LecunUniform**
LeCun uniform initializer.

It draws samples from a uniform distribution within [-limit, limit] where *limit* is *sqrt(3 / fan_in) [R3]* where *fan_in* is the number of input units in the weight matrix.

References

[R3]

class npdl.initialization.**GlorotUniform**
Glorot uniform initializer, also called Xavier uniform initializer.

It draws samples from a uniform distribution within [-limit, limit] where *limit* is *sqrt(6 / (fan_in + fan_out)) [R4]* where *fan_in* is the number of input units in the weight matrix and *fan_out* is the number of output units in the weight matrix.

References

[R4]

class npdl.initialization.**GlorotNormal**
Glorot normal initializer, also called Xavier normal initializer.

It draws samples from a truncated normal distribution centered on 0 with *stddev = sqrt(2 / (fan_in + fan_out)) [R5]* where *fan_in* is the number of input units in the weight matrix and *fan_out* is the number of output units in the weight matrix.

References

[R5]

class npdl.initialization.**HeNormal**
He normal initializer.

It draws samples from a truncated normal distribution centered on 0 with *stddev = sqrt(2 / fan_in) [R6]* where *fan_in* is the number of input units in the weight matrix.

References

[R6]

class npdl.initialization.**HeUniform**
He uniform variance scaling initializer.

It draws samples from a uniform distribution within [-limit, limit] where *limit* is *sqrt(6 / fan_in) [R7]* where *fan_in* is the number of input units in the weight matrix.

References

[R7]

class npdl.initialization.**Orthogonal**(*gain=1.0*)
Intialize weights as Orthogonal matrix.

Orthogonal matrix initialization *[R8]*. For n-dimensional shapes where n > 2, the n-1 trailing axes are flattened. For convolutional layers, this corresponds to the fan-in, so this makes the initialization usable for both dense and convolutional layers.

Parameters **gain** : float or 'relu'.

Scaling factor for the weights. Set this to `1.0` for linear and sigmoid units, to 'relu' or `sqrt(2)` for rectified linear units, and to `sqrt(2/(1+alpha**2))` for leaky rectified linear units with leakiness `alpha`. Other transfer functions may need different factors.

References

[R8]

npdl.initialization.**decompose_size**(*size*)
Computes the number of input and output units for a weight shape.

Parameters **size**

Integer shape tuple.

Returns A tuple of scalars, *(fan_in, fan_out)*.

npdl.objectives

Provides some minimal help with building loss expressions for training or validating a neural network.

These functions build element- or item-wise loss expressions from network predictions and targets.

Examples

Assuming you have a simple neural network for 3-way classification:

```
>>> import npdl
>>> model = npdl.model.Model()
>>> model.add(npdl.layers.Dense(n_out=100, n_in=50))
>>> model.add(npdl.layers.Dense(n_out=3,
>>>           activation=npdl.activation.Softmax()))
>>> model.compile(loss=npdl.objectives.SCCE(),
>>>           optimizer=npdl.optimizers.SGD(lr=0.005))
```

Objectives

BinaryCrossEntropy	Computes the binary cross-entropy between predictions and targets.
SoftmaxCategoricalCrossEntropy	Computes the categorical cross-entropy between predictions and targets.
MeanSquaredError	Computes the element-wise squared difference between two tensors.
HellingerDistance	Computes the multi-class hinge loss between predictions and targets.

Detailed Description

class npdl.objectives.**Objective**
An objective function (or loss function, or optimization score function) is one of the two parameters required to compile a model.

class npdl.objectives.**MeanSquaredError**
Computes the element-wise squared difference between two tensors.

$$L = (p - t)^2$$

Parameters **a, b** : Theano tensor

The tensors to compute the squared difference between.

Returns Theano tensor

An expression for the element-wise squared difference.

Notes

This is the loss function of choice for many regression problems or auto-encoders with linear output units.

npdl.objectives.**MSE**
alias of *MeanSquaredError*

class npdl.objectives.**HellingerDistance**
Computes the multi-class hinge loss between predictions and targets.

$$L_i = \max_{j \neq p_i}(0, t_j - t_{p_i} + \delta)$$

Parameters **predictions** : Theano 2D tensor

Predictions in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets : Theano 2D tensor or 1D tensor

Either a vector of int giving the correct class index per data point or a 2D tensor of one-hot encoding of the correct class in the same layout as predictions (non-binary targets in [0, 1] do not work!)

delta : scalar, default 1

The hinge loss margin

Returns Theano 1D tensor

An expression for the item-wise multi-class hinge loss

Notes

This is an alternative to the categorical cross-entropy loss for multi-class classification problems

npdl.objectives.**HeD**
alias of *HellingerDistance*

class npdl.objectives.**BinaryCrossEntropy**(*epsilon=1e-11*)
Computes the binary cross-entropy between predictions and targets.

$$L = -t \log(p) - (1 - t) \log(1 - p)$$

Returns Theano tensor

An expression for the element-wise binary cross-entropy.

Notes

This is the loss function of choice for binary classification problems and sigmoid output units.

backward(*outputs*, *targets*)
Backward pass.

Parameters **outputs** : numpy.array

Predictions in (0, 1), such as sigmoidal output of a neural network.

targets : numpy.array

Targets in [0, 1], such as ground truth labels.

forward(*outputs*, *targets*)
Forward pass.

Parameters **outputs** : numpy.array

Predictions in (0, 1), such as sigmoidal output of a neural network.

targets : numpy.array

Targets in [0, 1], such as ground truth labels.

npdl.objectives.**BCE**
alias of *BinaryCrossEntropy*

class npdl.objectives.**SoftmaxCategoricalCrossEntropy**(*epsilon=1e-11*)
Computes the categorical cross-entropy between predictions and targets.

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$

Parameters **predictions** : Theano 2D tensor

Predictions in (0, 1), such as softmax output of a neural network, with data points in rows and class probabilities in columns.

targets : Theano 2D tensor or 1D tensor

Either targets in [0, 1] matching the layout of *predictions*, or a vector of int giving the correct class index per data point.

Returns Theano 1D tensor

An expression for the item-wise categorical cross-entropy.

Notes

This is the loss function of choice for multi-class classification problems and softmax output units. For hard targets, i.e., targets that assign all of the probability to a single class per data point, providing a vector of int for the targets is usually slightly more efficient than providing a matrix with a single 1.0 per row.

npdl.objectives.**SCCE**
alias of *SoftmaxCategoricalCrossEntropy*

npdl.optimizers

Functions to generate Theano update dictionaries for training.

The update functions implement different methods to control the learning rate for use with stochastic gradient descent.

Update functions take a loss expression or a list of gradient expressions and a list of parameters as input and return an ordered dictionary of updates:

Examples

Using *SGD* to define an update dictionary for a toy example network:

```
>>> import npdl
>>> from npdl.activation import ReLU
>>> from npdl.activation import Softmax
>>> from npdl.objectives import SCCE
>>> model = npdl.model.Model()
>>> model.add(npdl.layers.Dense(n_out=100, n_in=50, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=200, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=100, activation=ReLU()))
>>> model.add(npdl.layers.Dense(n_out=10, activation=Softmax()))
>>> model.compile(loss=SCCE(), optimizer=npdl.optimizers.SGD(lr=0.005))
```

Optimizers

SGD	Stochastic Gradient Descent (SGD) updates
Momentum	Stochastic Gradient Descent (SGD) updates with momentum
NesterovMomentum	Stochastic Gradient Descent (SGD) updates with Nesterov momentum
Adagrad	Adagrad updates
RMSprop	RMSProp updates
Adadelta	Adadelta updates
Adam	Adam updates
Adamax	Adamax updates

Detailed Description

class npdl.optimizers.**SGD**(*lr=0.001*, *clip=-1*)
Stochastic Gradient Descent (SGD) updates

Generates update expressions of the form:

- `param := param - learning_rate * gradient`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

Returns OrderedDict

A dictionary mapping each parameter to its update expression

class npdl.optimizers.**Momentum**(*lr=0.01*, *momentum=0.9*)
Stochastic Gradient Descent (SGD) updates with momentum

Generates update expressions of the form:

- `velocity := momentum * velocity - learning_rate * gradient`
- `param := param + velocity`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

See also:

apply_momentum Generic function applying momentum to updates

nesterov_momentum Nesterov's variant of SGD with momentum

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

class npdl.optimizers.**NesterovMomentum**

Stochastic Gradient Descent (SGD) updates with Nesterov momentum

Generates update expressions of the form:

- `velocity := momentum * velocity - learning_rate * gradient`
- `param := param + momentum * velocity - learning_rate * gradient`

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

momentum : float or symbolic scalar, optional

The amount of momentum to apply. Higher momentum results in smoothing over more update steps. Defaults to 0.9.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

See also:

apply_nesterov_momentum Function applying momentum to updates

Notes

Higher momentum also results in larger update steps. To counter that, you can optionally scale your learning rate by *1 - momentum*.

The classic formulation of Nesterov momentum (or Nesterov accelerated gradient) requires the gradient to be evaluated at the predicted next position in parameter space. Here, we use the formulation described at https://github.com/lisa-lab/pylearn2/pull/136#issuecomment-10381617, which allows the gradient to be evaluated at the current parameters.

class npdl.optimizers.**Adagrad**

Adagrad updates

Scale learning rates by dividing with the square root of accumulated squared gradients. See *[R15]* for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

Using step size eta Adagrad calculates the learning rate for feature i at time step t as:

$$\eta_{t,i} = \frac{\eta}{\sqrt{\sum_{t'}^{t} g_{t',i}^2 + \epsilon}} g_{t,i}$$

as such the learning rate is monotonically decreasing.

Epsilon is not included in the typical formula, see *[R16]*.

References

[R15], *[R16]*

class npdl.optimizers.**RMSprop**

RMSProp updates

Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients. See *[R17]* for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

rho : float or symbolic scalar

Gradient moving average decay factor

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

rho should be between 0 and 1. A value of *rho* close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

Using the step size η and a decay factor ρ the learning rate η_t is calculated as:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \frac{\eta}{\sqrt{r_t + \epsilon}}$$

References

[R17]

class npdl.optimizers.**Adadelta**

Adadelta updates

Scale learning rates by the ratio of accumulated gradients to accumulated updates, see *[R18]* and notes for further description.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

The learning rate controlling the size of update steps

rho : float or symbolic scalar

Squared gradient moving average decay factor

epsilon : float or symbolic scalar

Small value added for numerical stability

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

rho should be between 0 and 1. A value of rho close to 1 will decay the moving average slowly and a value close to 0 will decay the moving average fast.

rho = 0.95 and epsilon=1e-6 are suggested in the paper and reported to work for multiple datasets (MNIST, speech).

In the paper, no learning rate is considered (so learning_rate=1.0). Probably best to keep it at this value. epsilon is important for the very first update (so the numerator does not become 0).

Using the step size eta and a decay factor rho the learning rate is calculated as:

$$r_t = \rho r_{t-1} + (1-\rho) * g^2$$
$$\eta_t = \eta \frac{\sqrt{s_{t-1} + \epsilon}}{\sqrt{r_t + \epsilon}}$$
$$s_t = \rho s_{t-1} + (1-\rho) * (\eta_t * g)^2$$

References

[R18]

class npdl.optimizers.**Adam**

Adam updates

Adam updates implemented as in *[R19]*.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

Learning rate

beta1 : float or symbolic scalar

Exponential decay rate for the first moment estimates.

beta2 : float or symbolic scalar

Exponential decay rate for the second moment estimates.

epsilon : float or symbolic scalar

Constant for numerical stability.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

Notes

The paper *[R19]* includes an additional hyperparameter lambda. This is only needed to prove convergence of the algorithm and has no practical use (personal communication with the authors), it is therefore omitted here.

References

[R19]

class npdl.optimizers.**Adamax**

Adamax updates

Adamax updates implemented as in *[R20]*. This is a variant of of the Adam algorithm based on the infinity norm.

Parameters **loss_or_grads** : symbolic expression or list of expressions

A scalar loss expression, or a list of gradient expressions

params : list of shared variables

The variables to generate update expressions for

learning_rate : float or symbolic scalar

Learning rate

beta1 : float or symbolic scalar

Exponential decay rate for the first moment estimates.

beta2 : float or symbolic scalar

Exponential decay rate for the weighted infinity norm estimates.

epsilon : float or symbolic scalar

Constant for numerical stability.

Returns OrderedDict

A dictionary mapping each parameter to its update expression

References

[R20]

npdl.model

Linear stack of layers.

Detailed Description

class npdl.model.**Model**(*layers=None*)

predict(*X*)
Calculate an output Y for the given input X.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[R2] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.” arXiv preprint arXiv:1312.6120 (2013).

[R3] LeCun 98, Efficient Backprop, http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[R4] Glorot & Bengio, AISTATS 2010. http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

[R5] Glorot & Bengio, AISTATS 2010. http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

[R6] He et al., http://arxiv.org/abs/1502.01852

[R7] He et al., http://arxiv.org/abs/1502.01852

[R8] Saxe, Andrew M., James L. McClelland, and Surya Ganguli. “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks.” arXiv preprint arXiv:1312.6120 (2013).

[R15] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12:2121-2159.

[R16] Chris Dyer: Notes on AdaGrad. http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf

[R17] Tieleman, T. and Hinton, G. (2012): Neural Networks for Machine Learning, Lecture 6.5 - rmsprop. Coursera. http://www.youtube.com/watch?v=O3sxAc4hxZU (formula @5:20)

[R18] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.

[R19] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[R20] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

Python Module Index

n

Index

P

R

S

T

U

Z