numpydl Documentation

Release 0.4.0

NumpyDL

Jun 18, 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:

1. *Pure* in Numpy
2. *Native* to Python
3. *Automatic differentiations* are basically supported
4. *Commonly used models* are provided: MLP, RNNs, LSTMs and CNNs
5. *API* like `Keras` library
6. *Examples* for several AI tasks
7. *Application* for a toy chatbot

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.

1.1 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.

1.1.1 Prerequisites

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`.

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.

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.

1.1.2 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`.

1.1.3 Development installation

install from source

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.

contribute

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*!

1.2 Development

The NumpyDL project is started by Chao-Ming Wang in February 2017. It is developed by a core team of five people: Chao-Ming Wang, Jiao-Mei Liu, Shu-Ting Kang, Xiao-Xuan Cui, Jin-Ze Li on Github: https://github.com/oujago/NumpyDL . The goal of NumpyDL is making `deep learning` easy to learn and easy to use in native Numpy.

As an open-source project, we highly welcome contributions! Every bit helps and will be credited!

1.2.1 Philosophy

The development of NumpyDl is guided by a number of design goals:

- **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.

1.2.2 What to contribute

Give feedback

To send us general feedback, questions or ideas for improvement, please post on issue tracker on GitHub. Or, you can directly e-mail to Chao-Ming Wang's Email.

If you have a very concrete feature proposal, add it to the issue tracker on GitHub:

- Explain how it would work, and link to a scientific paper if applicable.
- Keep the scope as narrow as possible, to make it easier to implement.

Fix bugs

Look through the GitHub issues for bug reports. Anything tagged with "bug" is open to whoever wants to implement it. If you discover a bug in NumpyDL you can fix yourself, by all means feel free to just implement a fix and not report it first.

Implement features

Look through the GitHub issues for feature proposals. Anything tagged with "feature" or "enhancement" is open to whoever wants to implement it. If you have a feature in mind you want to implement yourself, please note that Lasagne has a fairly narrow focus and we strictly follow a set of *design principles*, so we cannot guarantee upfront that your code will be included. Please do not hesitate to just propose your idea in a GitHub issue first, so we can discuss it and/or guide you through the implementation.

Write documentation

Whenever you find something not explained well, misleading, glossed over or just wrong, please update it! The *Edit on GitHub* link on the top right of every documentation page and the *[source]* link for every documented entity in the API reference will help you to quickly locate the origin of any text.

Write tutorial

How to combine our Numpy code with live examples and detailed explanations about deep learning is NumpyDL's ultimate goals. So, please contribute your good ideas about how to make good tutorials.

1.2.3 How to contribute

Edit on GitHub

As a very easy way of just fixing issues in the documentation, use the *Edit on GitHub* link on the top right of a documentation page or the *[source]* link of an entity in the API reference to open the corresponding source file in GitHub, then click the *Edit this file* link to edit the file in your browser and send us a Pull Request. All you need for this is a free GitHub account.

For any more substantial changes, please follow the steps below to setup NumpyDL for development.

Development setup

First, follow the instructions for performing a development installation of NumpyDL (including forking on GitHub): *Development installation*

To be able to run the tests and build the documentation locally, install additional requirements with: `pip install -r requirements-dev.txt` (adding `--user` if you want to install to your home directory instead).

Documentation

The documentation is generated with Sphinx. To build it locally, run the following commands:

```
python setup.py install
cd docs
make html
```

Afterwards, open `docs/_build/html/index.html` to view the documentation as it would appear on readthedocs. If you changed a lot and seem to get misleading error messages or warnings, run `make clean html` to force Sphinx to recreate all files from scratch.

When writing docstrings, follow existing documentation as much as possible to ensure consistency throughout the library. For additional information on the syntax and conventions used, please refer to the following documents:

- reStructuredText Primer
- Sphinx reST markup constructs
- A Guide to NumPy/SciPy Documentation

Testing

NumpyDL has a code coverage of 100%, which has proven very helpful in the past, but also creates some duties:

- Whenever you change any code, you should test whether it breaks existing features by just running the test suite. The test suite will also be run by Travis for any Pull Request to NumpyDL.
- Any code you add needs to be accompanied by tests ensuring that nobody else breaks it in future. Coveralls will check whether the code coverage stays at 100% for any Pull Request to NumpyDL.
- Every bug you fix indicates a missing test case, so a proposed bug fix should come with a new test that fails without your fix.

To run the full test suite, just do

```
py.test
```

Testing will take over several minutes for running for there are example testing. It will end with a code coverage report specifying which code lines are not covered by tests, if any. Furthermore, it will list any failed tests, and failed PEP8 checks.

To only run tests matching a certain name pattern, use the `-k` command line switch, e.g., `-k pool` will run the pooling layer tests only.

To land in a `pdb` debug prompt on a failure to inspect it more closely, use the `--pdb` switch.

Finally, for a loop-on-failing mode, do `pip install pytest-xdist` and run `py.test -f`. This will pause after the run, wait for any source file to change and run all previously failing tests again.

Sending Pull Requests

When you're satisfied with your addition, the tests pass and the documentation looks good without any markup errors, commit your changes to a new branch, push that branch to your fork and send us a Pull Request via GitHub's web interface.

All these steps are nicely explained on GitHub: https://guides.github.com/introduction/flow/

When filing your Pull Request, please include a description of what it does, to help us reviewing it. If it is fixing an open issue, say, issue #123, add *Fixes #123*, *Resolves #123* or *Closes #123* to the description text, so GitHub will close it when your request is merged.

CHAPTER 2

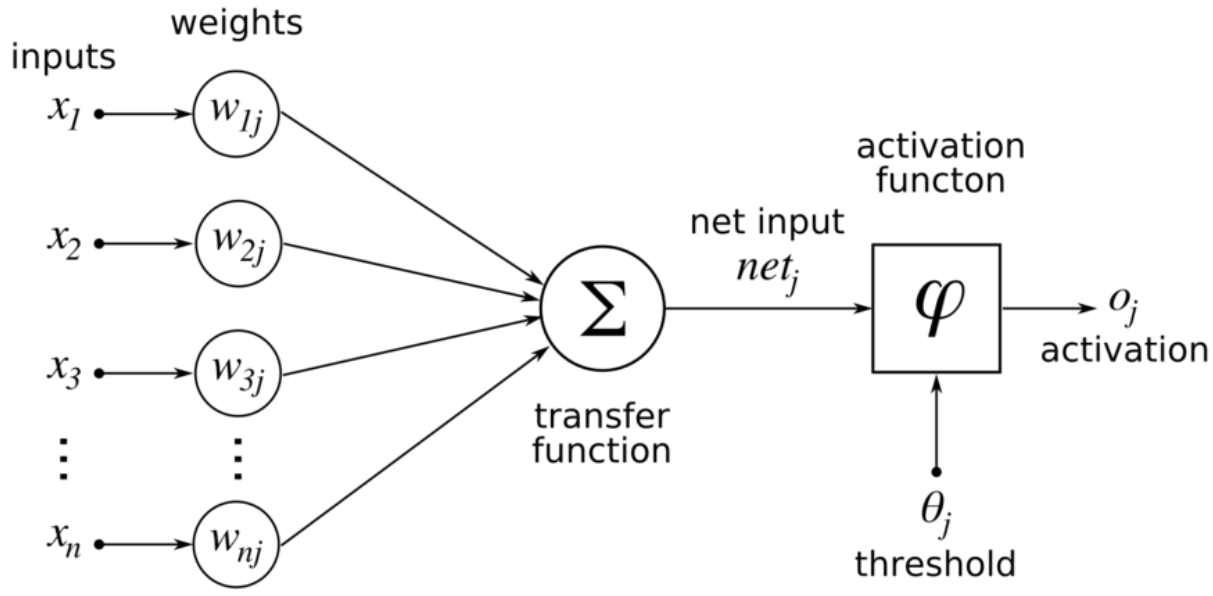
Tutorials

This is the tutorials of Deep Learning.

2.1 Activation

2.1.1 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.



2.1.2 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:

$$M = M_1 M_2 \cdots M_n$$

2.1.3 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:

Sigmoid

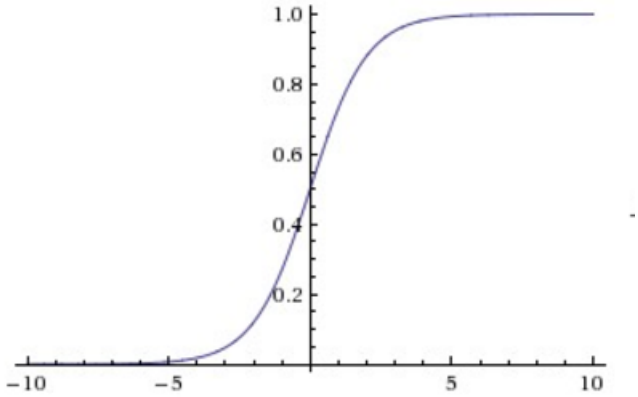


Fig. 2.1: Sigmoid non-linearity squashes real numbers to range between $[0, 1]$.

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.

Tangent

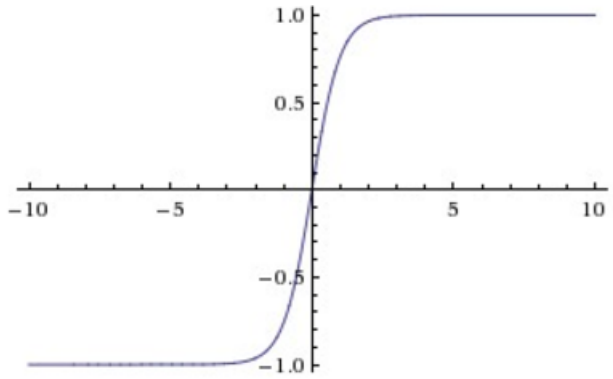


Fig. 2.2: The tanh non-linearity squashes real numbers to range between $[-1, 1]$.

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:

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

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.

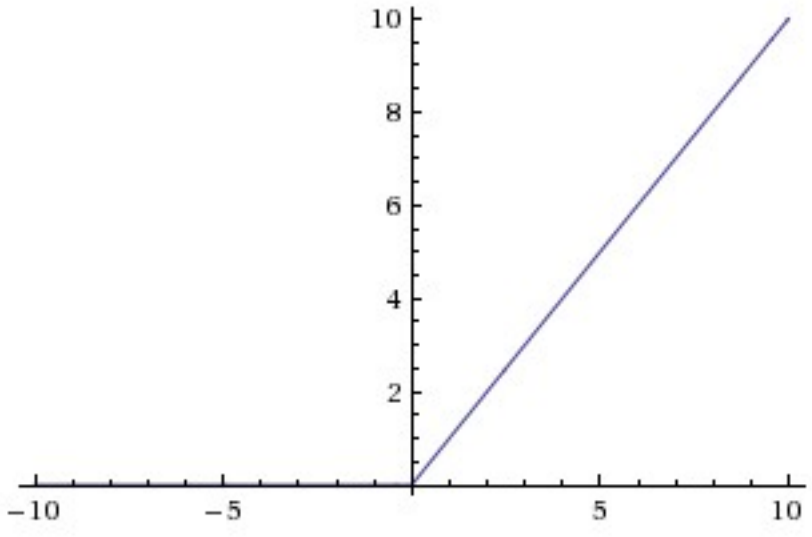


Fig. 2.3: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$.

- (-) 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.

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.

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).

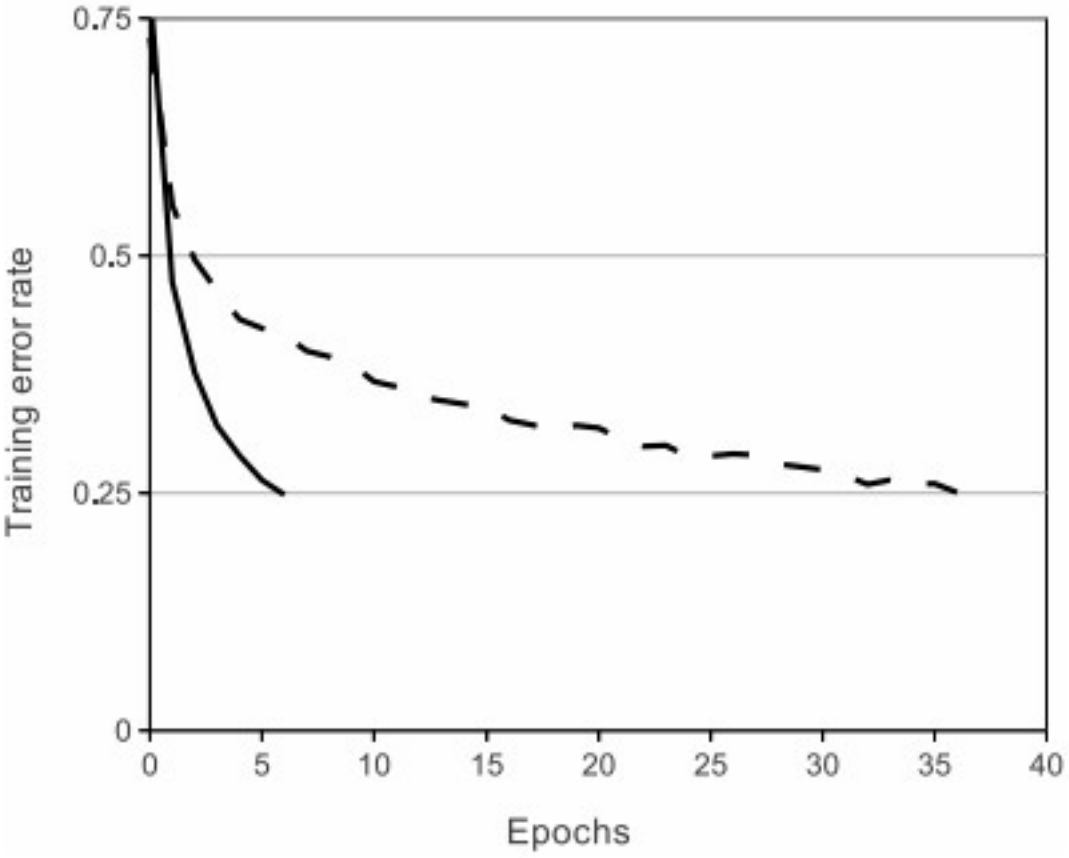
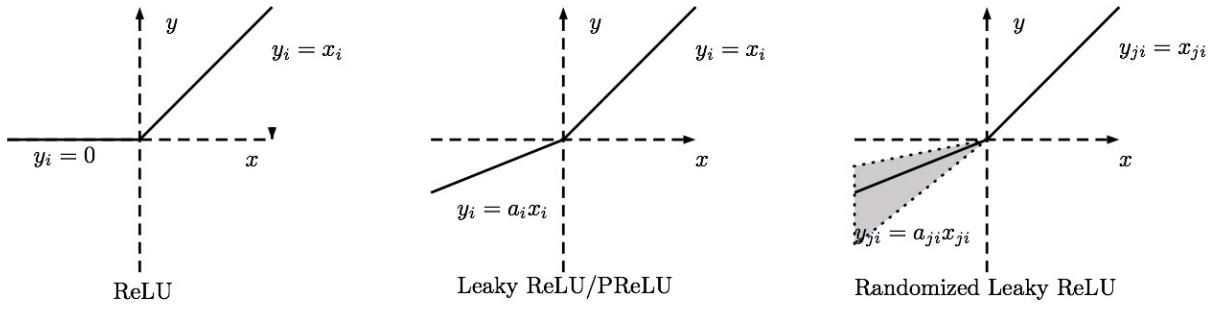


Fig. 2.4: A plot from Krizhevsky et al.[1] paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.



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.

2.1.4 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.

2.2 Initialization

2.2.1 Introduction

As we all know, the solution to a non-convex optimization algorithm (like stochastic gradient descent) depends on the initial values of the parameters. This post is about choosing initialization parameters for deep networks and how it affects the convergence. We will also discuss the related topic of vanishing gradients.

First, let's go back to the time of sigmoidal activation functions and initialization of parameters using IID Gaussian or uniform distributions with fairly arbitrarily set variances. Building deep networks was difficult because of exploding or vanishing activations and gradients. Let's take activations first: If all your parameters are too small, the variance of your activations will drop in each layer. This is a problem if your activation function is sigmoidal, since it is approximately linear close to 0. That is, you gradually lose your non-linearity, which means there is no benefit to having multiple layers. If, on the other hand, your activations become larger and larger, then your activations will saturate and become meaningless, with gradients approaching 0.

Let us consider one layer and forget about the bias. Note that the following analysis and conclusion is taken from Glorot and Bengio[1]. Consider a weight matrix $W \in R^{mn}$, where each element was drawn from an IID Guassian with variance $Var(W)$. Note that we are a bit abusive with notation letting W denote both a matrix and a univariate random variable. We also assume there is no correlation between our input and our weights and both are zero-mean. If we consider one filter (row) in W, say W (a random vector), then the variance of the output signal over the input signal is:

$$\frac{Var(W^T x)}{Var(x)} = \frac{\sum_n^m Var(W_n x_n)}{Var(x)} = \frac{nVar(W)Var(x)}{Var(x)} = nVar(W)$$

As we build a deep network, we want the variance of the signal going forward in the network to remain the same, thus it would be advantageous if $nVar(W) = 1$. The same argument can be made for the gradients, the signal going backward in the network, and the conclusion is that we would also like $mVar(W) = 1$. Unless $n = m$, it is impossible to sastify both of these conditions. In practice, it works well if both are approximately satisfied. One thing

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in International conference on artificial intelligence and statistics, 2010, pp. 249–256.

that has never been clear to me is why it is only necessary to satisfy these conditions when picking the initialization values of W. It would seem that we have no guarantee that the conditions will remain true as the network is trained.

Nevertheless, this *Xavier initialization* (after Glorot's first name) is a neat trick that works well in practice. However, along came rectified linear units (ReLU), a non-linearity that is scale-invariant around 0 and does not saturate at large input values. This seemingly solved both of the problems the sigmoid function had; or were they just alleviated? I am unsure of how widely used Xavier initialization is, but if it is not, perhaps it is because ReLU seemingly eliminated this problem.

However, take the most competative network as of recently, VGG[2]. They do not use this kind of initialization, although they report that it was tricky to get their networks to converge. They say that they first trained their most shallow architecture and then used that to help initialize the second one, and so forth. They presented 6 networks, so it seems like an awfully complicated training process to get to the deepest one.

A recent paper by He et al.[3] presents a pretty straightforward generalization of ReLU and Leaky ReLU. What is more interesting is their emphasis on the benefits of Xavier initialization even for ReLU. They re-did the derivations for ReLUs and discovered that the conditions were the same up to a factor 2. The difficulty Simonyan and Zisserman had training VGG is apparently avoidable, simply by using Xavier intialization (or better yet the ReLU adjusted version). Using this technique, He et al. reportedly trained a whopping 30-layer deep network to convergence in one go.

Another recent paper tackling the signal scaling problem is by Ioffe and Szegedy[4]. They call the change in scale internal covariate shift and claim this forces learning rates to be unnecessarily small. They suggest that if all layers have the same scale and remain so throughout training, a much higher learning rate becomes practically viable. You cannot just standardize the signals, since you would lose expressive power (the bias disappears and in the case of sigmoids we would be constrained to the linear regime). They solve this by re-introducing two parameters per layer, scaling and bias, added again after standardization. The training reportedly becomes about 6 times faster and they present state-of-the-art results on ImageNet. However, I'm not certain this is the solution that will stick.

I reckon we will see a lot more work on this frontier in the next few years. Especially since it also relates to the – right now wildly popular – Recurrent Neural Network (RNN), which connects output signals back as inputs. The way you train such network is that you unroll the time axis, treating the result as an extremely deep feedforward network. This greatly exacerbates the vanishing gradient problem. A popular solution, called Long Short-Term Memory (LSTM), is to introduce memory cells, which are a type of teleport that allows a signal to jump ahead many time steps. This means that the gradient is retained for all those time steps and can be propagated back to a much earlier time without vanishing.

2.2.2 Xavier Initialization

Why's Xavier initialization important?

In short, it helps signals reach deep into the network.

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it's too tiny to be useful.
- If the weights in a network start too large, then the signal grows as it passes through each layer until it's too massive to be useful.

Xavier initialization makes sure the weights are 'just right', keeping the signal in a reasonable range of values through many layers.

To go any further than this, you're going to need a small amount of statistics - specifically you need to know about random distributions and their variance.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," arXiv:1502.01852 [cs], Feb. 2015.
[4] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv:1502.03167 [cs], Feb. 2015.

What's Xavier initialization?

For specific implementation, it's initializing the weights in your network by drawing them from a distribution with zero mean and a specific variance,

$$Var(W) = \frac{1}{n_{in}}$$

where W is the initialization distribution for the neuron in question, and n_{in} is the number of neurons feeding into it. The distribution used is typically Gaussian or uniform.

It's worth mentioning that Glorot & Bengio's paper[1] originally recommended using:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

where n_{out} is the number of neurons the result is fed to.

Where did those formulas come from?

Suppose we have an input X with n components and a linear neuron with random weights W that spits out a number Y. What's the variance of Y? Well, we can write

$$Y = W_1X_1 + W_2X_2 + +W_nX_n$$

And from Wikipedia[5] we can work out that W_iX_i is going to have variance

$$Var(W_iX_i) = E[X_i]^2Var(W_i) + E[W_i]^2Var(X_i) + Var(W_i)Var(i_i)$$

Now if our inputs and weights both have mean 0, that simplifies to

$$Var(W_iX_i) = Var(W_i)Var(X_i)$$

Then if we make a further assumption that the X_i and W_i are all independent and identically distributed, we can work out that the variance of Y is[6]

$$Var(Y) = Var(W_1X_1 + W_2X_2 + +W_nX_n) = nVar(W_i)Var(X_i)$$

Or in words: the variance of the output is the variance of the input, but scaled by $nVar(W_i)$. So if we want the variance of the input and output to be the same, that means $nVar(W_i)$ should be 1. Which means the variance of the weights should be

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

Voila. There's your Xavier initialization.

Glorot & Bengio's formula needs a tiny bit more work. If you go through the same steps for the backpropagated signal, you find that you need

$$Var(W_i) = \frac{1}{n_{out}}$$

to keep the variance of the input gradient & the output gradient the same. These two constraints can only be satisfied simultaneously if $n_{in} = n_{out}$, so as a compromise, Glorot & Bengio take the average of the two:

$$Var(W_i) = \frac{2}{n_{in} + n_{out}}$$

Caffe authors used the n_{in}-only variant. The two possibilities that come to mind are:

[5] https://en.wikipedia.org/wiki/Variance#Product_of_independent_variables
[6] https://en.wikipedia.org/wiki/Variance#Sum_of_uncorrelated_variables_.28Bienaym.C3.A9_formula.29

- that preserving the forward-propagated signal is much more important than preserving the back-propagated one.
- that for implementation reasons, it's a pain to find out how many neurons in the next layer consume the output of the current one.

It is. But it works. Xavier initialization was one of the big enablers of the move away from per-layer generative pre-training.

2.3 Objective

2.3.1 What is the Objective Function?

The objective of a linear programming problem will be to maximize or to minimize some numerical value. This value may be the expected net present value of a project or a forest property; or it may be the cost of a project; it could also be the amount of wood produced, the expected number of visitor-days at a park, the number of endangered species that will be saved, or the amount of a particular type of habitat to be maintained. Linear programming is an extremely general technique, and its applications are limited mainly by our imaginations and our ingenuity.

The objective function indicates how much each variable contributes to the value to be optimized in the problem. The objective function takes the following general form:

$$maximize or$$
$$minimize Z = \sum_{i=1}^{n} c_i X_i$$

where

- c_i = the objective function coefficient corresponding to the ith variable
- X_i = the i-th decision variable.
- The summation notation used here was discussed in the section above on linear functions. The summation notation for the objective function can be expanded out as follows: $Z = \sum_{i=1}^{n} c_i X_i = c_1 X_1 + c_2 X_2 + \cdots + c_n X_n$

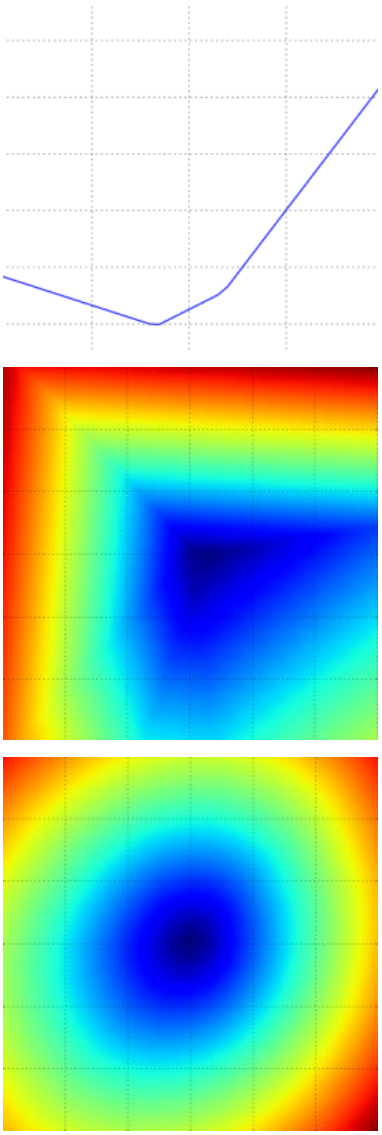
The coefficients of the objective function indicate the contribution to the value of the objective function of one unit of the corresponding variable. For example, if the objective function is to maximize the present value of a project, and X_i is the i-th possible activity in the project, then c_i (the objective function coefficient corresponding to X_i) gives the net present value generated by one unit of activity i. As another example, if the problem is to minimize the cost of achieving some goal, X_i might be the amount of resource i used in achieving the goal. In this case, c_i would be the cost of using one unit of resource i.

Note that the way the general objective function above has been written implies that each variable has a coefficient in the objective function. Of course, some variables may not contribute to the objective function. In this case, you can either think of the variable as having a coefficient of zero, or you can think of the variable as not being in the objective function at all.

2.3.2 Visualizing the Objective function

The loss functions we'll look at in this class are usually defined over very high-dimensional spaces (e.g. in CIFAR-10 a linear classifier weight matrix is of size $[10x3073]$ for a total of 30,730 parameters), making them difficult to visualize. However, we can still gain some intuitions about one by slicing through the high-dimensional space along rays (1 dimension), or along planes (2 dimensions). For example, we can generate a random weight matrix W (which corresponds to a single point in the space), then march along a ray and record the loss function value along the way. That is, we can generate a random direction W_1 and compute the loss along this direction by evaluating $L(W + aW_1)$ for different values of a. This process generates a simple plot with the value of a as the x-axis and the value of the

loss function as the y-axis. We can also carry out the same procedure with two dimensions by evaluating the loss $L(W + aW_1 + bW_2)$ as we vary a, b. In a plot, a, b could then correspond to the x-axis and the y-axis, and the value of the loss function can be visualized with a color:



Loss function landscape for the Multiclass SVM (without regularization) for one single example (left,middle) and for a hundred examples (right) in CIFAR-10. **Left**: one-dimensional loss by only varying a. **Middle**, **Right**: two-dimensional loss slice, Blue = low loss, Red = high loss. Notice the piecewise-linear structure of the loss function. The losses for multiple examples are combined with average, so the bowl shape on the right is the average of many piece-wise linear bowls (such as the one in the middle).

We can explain the piecewise-linear structure of the loss function by examining the math. For a single example we have:

$$L_i = \sum_{j \neq y_i} [max(0, w_j^T x_i w_{y_i}^T x_i + 1)]$$

It is clear from the equation that the data loss for each example is a sum of (zero-thresholded due to the $max(0,)$ function) linear functions of W. Moreover, each row of W (i.e. w_j) sometimes has a positive sign in front of it (when it corresponds to a wrong class for an example), and sometimes a negative sign (when it corresponds to the correct class for that example). To make this more explicit, consider a simple dataset that contains three 1-dimensional points

and three classes. The full SVM loss (without regularization) becomes:

$$
\begin{aligned}
L_0 &= max(0, w_1^T x_0 w_0^T x_0 + 1) + max(0, w_2^T x_0 w_0^T x^0 + 1) \\
L_1 &= max(0, w_0^T x_1 w_1^T x_1 + 1) + max(0, w_2^T x_1 w_1^T x_1 + 1) \\
L_2 &= max(0, w_0^T x_2 w_2^T x_2 + 1) + max(0, w_1^T x_2 w_2^T x_2 + 1) \\
L &= (L_0 + L_1 + L_2)/3
\end{aligned}
$$

Since these examples are 1-dimensional, the data x_i and weights w_j are numbers. Looking at, for instance, w_0, some terms above are linear functions of w_0 and each is clamped at zero. We can visualize this as follows:

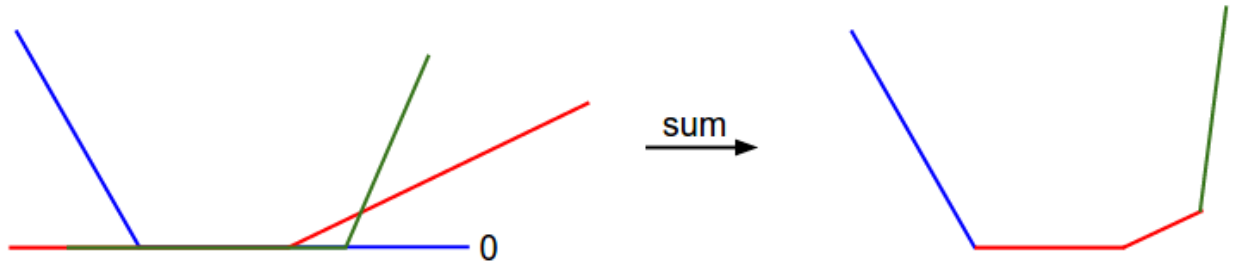


Fig. 2.5: 1-dimensional illustration of the data loss. The x-axis is a single weight and the y-axis is the loss. The data loss is a sum of multiple terms, each of which is either independent of a particular weight, or a linear function of it that is thresholded at zero. The full SVM data loss is a 30,730-dimensional version of this shape.

As an aside, you may have guessed from its bowl-shaped appearance that the SVM cost function is an example of a convex function[1]. There is a large amount of literature devoted to efficiently minimizing these types of functions, and you can also take a Stanford class on the topic (convex optimization[2]). Once we extend our score functions ff to Neural Networks our objective functions will become non-convex, and the visualizations above will not feature bowls but complex, bumpy terrains.

Non-differentiable loss functions. As a technical note, you can also see that the kinks in the loss function (due to the max operation) technically make the loss function non-differentiable because at these kinks the gradient is not defined. However, the subgradient still exists and is commonly used instead. In this class will use the terms subgradient[3] and gradient interchangeably.

2.4 Optimizer

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent (e.g. lasagne's[25], caffe's[26], and keras'[27] documentation). These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

This blog post aims at providing you with intuitions towards the behaviour of different algorithms for optimizing gradient descent that will help you put them to use. We are first going to look at the different variants of gradient descent. We will then briefly summarize challenges during training. Subsequently, we will introduce the most common optimization algorithms by showing their motivation to resolve these challenges and how this leads to the derivation

[1] https://en.wikipedia.org/wiki/Convex_function
[2] http://stanford.edu/~boyd/cvxbook/
[3] https://en.wikipedia.org/wiki/Subderivative
[25] http://lasagne.readthedocs.org/en/latest/modules/updates.html
[26] http://caffe.berkeleyvision.org/tutorial/solver.html
[27] http://keras.io/optimizers/

of their update rules. We will also take a short look at algorithms and architectures to optimize gradient descent in a parallel and distributed setting. Finally, we will consider additional strategies that are helpful for optimizing gradient descent.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\bigtriangledown_\theta J(\theta)$ w.r.t. to the parameters. The learning rate $\eta\eta$ determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley. If you are unfamiliar with gradient descent, you can find a good introduction on optimizing neural networks here[28].

2.4.1 Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

Batch gradient descent

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta * \bigtriangledown_\theta J(\theta)$$

As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory. Batch gradient descent also doesn't allow us to update our model online, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

For a pre-defined number of epochs, we first compute the gradient vector `params_grad` of the loss function for the whole dataset w.r.t. our parameter vector `params`. Note that state-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters. If you derive the gradients yourself, then gradient checking is a good idea. (See here for some great tips on how to check gradients properly.)

We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta * \bigtriangledown_\theta J(\theta; x^{(i)}; y^{(i)})$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

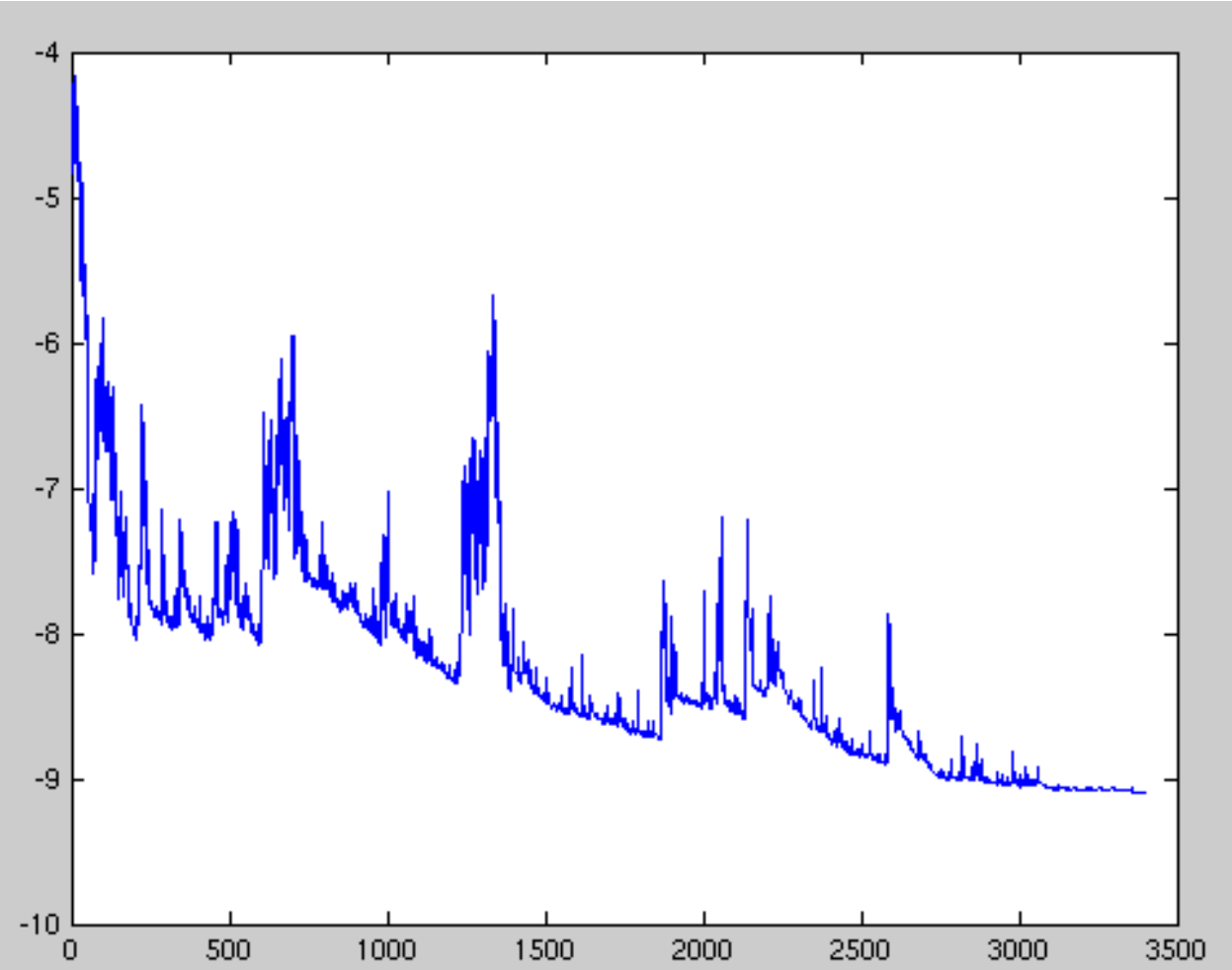


Fig. 2.6: Image 1: SGD fluctuation.

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image 1.

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch as explained in *Shuffling and Curriculum Learning*.

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta * \bigtriangledown_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. Note: In modifications of SGD in the rest of this post, we leave out the parameters $x^{(i:i+n)}; y^{(i:i+n)}$ for simplicity.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

2.4.2 Challenges

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Learning rate schedules[11] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a

[28] http://cs231n.github.io/optimization-1/

[11] H. Robinds and S. Monro, "A stochastic approximation method," Annals of Mathematical Statistics, vol. 22, pp. 400–407, 1951.

threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics[10].

- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al.[19] argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Gradient descent optimization algorithms

In the following, we will outline some algorithms that are widely used by the deep learning community to deal with the aforementioned challenges. We will not discuss algorithms that are infeasible to compute in practice for high-dimensional data sets, e.g. second-order methods such as Newton's method[29].

Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another[1], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Image 2.

Fig. 2.7: Image 2: SGD without momentum

Fig. 2.8: Image 3: SGD with momentum

Momentum[2] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \gamma \bigtriangledown_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Note: Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

[10] Darken, C., Chang, J., & Moody, J. (1992). Learning rate schedules for faster stochastic gradient search. Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop, (September), 1–11. http://doi.org/10.1109/NNSP.1992.253713

[19] Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14. Retrieved from http://arxiv.org/abs/1406.2572

[29] https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

[1] Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society.

[2] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6

Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We'd like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

Nesterov accelerated gradient (NAG)[7] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term γv_{t1} to move the parameters 0. Computing $\theta - \gamma v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters $\theta\theta$ but w.r.t. the approximate future position of our parameters:

$$v_t = \gamma v_{t-1} + \eta \bigtriangledown_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Again, we set the momentum term $\gamma\gamma$ to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Image 4) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (red vector), which results in the complete NAG update (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks[8].

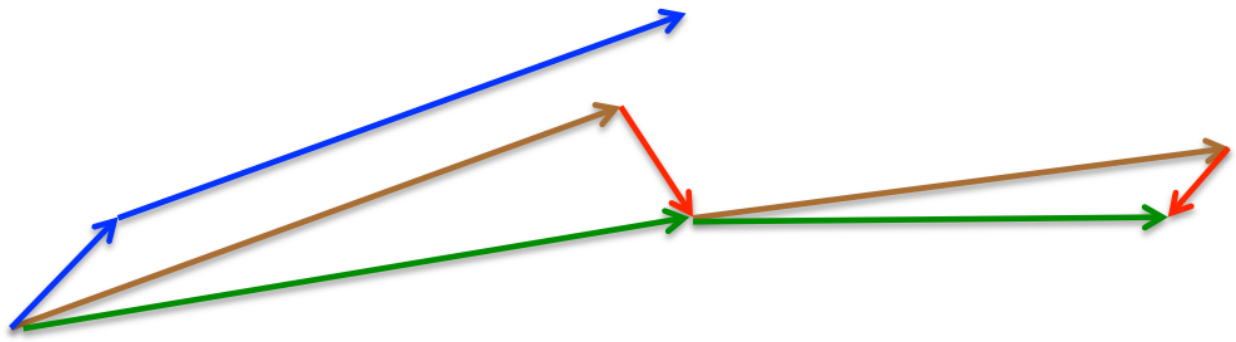


Fig. 2.9: Image 4: Nesterov update

Refer to here for another explanation about the intuitions behind NAG, while Ilya Sutskever gives a more detailed overview in his PhD thesis[9].

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

Adagrad

Adagrad[3] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is

[7] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

[8] Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2012). Advances in Optimizing Recurrent Networks. Retrieved from http://arxiv.org/abs/1212.0901

[9] Sutskever, I. (2013). Training Recurrent neural Networks. PhD Thesis.

[3] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html

well-suited for dealing with sparse data. Dean et al.[4] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which – among other things – learned to recognize cats in Youtube videos[30]. Moreover, Pennington et al.[5] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter $_i$ at time step tt:

$$g_{t,i} = \bigtriangledown_\theta J(\theta_i)$$

The SGD update for every parameter $_i$ at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i}\eta * g_{t,i}$$

In its update rule, Adagrad modifies the general learning rate $\eta\eta$ at each time step t for every parameter $_i$ based on the past gradients that have been computed for $_i$:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} * g_{t,i}$$

$G_t \in R^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. $_i$ up to time step t[24], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e8$). Interestingly, without the square root operation, the algorithm performs much worse.

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\bigodot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \bigodot g_t$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

Adadelta

Adadelta[6] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing ww previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step tt then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

[4] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, … Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, 1–11. http://doi.org/10.1109/ICDAR.2011.95

[30] https://www.wired.com/2012/06/google-x-neural-network/

[5] Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 1532–1543. http://doi.org/10.3115/v1/D14-1162

[24] Duchi et al. [3] give this matrix as an alternative to the full matrix containing the outer products of all previous gradients, as the computation of the matrix square root is infeasible even for a moderate number of parameters dd.

[6] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\triangle\theta_t$:

$$\triangle \theta_t = -\eta * g_{t,i}$$
$$\theta_{t+1} = \theta_t + \triangle\theta_t$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\triangle\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \bigodot g_t$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\triangle\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \bigodot g_t$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\triangle\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\triangle\theta^2] = \gamma E[\triangle\theta^2]_{t-1} + (1-\gamma) \triangle \theta^2$$

The root mean squared error of parameter updates is thus:

$$RMS[\triangle\theta]_t = \sqrt{E(\triangle\theta^2)_t + \epsilon}$$

Since $RMS[\triangle\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\triangle\theta]_{t-1}$ finally yields the Adadelta update rule:

$$\triangle \theta_t = -\frac{RMS[\triangle\theta]_{t-1}}{RMS[\triangle\theta]_t} g_t$$
$$\theta_{t+1} = \theta_t + \triangle\theta_t$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class[31].

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

[31] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam

Adaptive Moment Estimation (Adam)[15] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t1} + (1\beta_1) g_t$$
$$v_t = \beta_2 v_t 1 + (1\beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$
$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m_t}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 108 for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Visualization of algorithms

The following two animations provide some intuitions towards the optimization behaviour of the presented optimization algorithms. Also have a look here[32] for a description of the same images by Karpathy and another concise overview of the algorithms discussed.

Fig. 2.10: Image 5: SGD optimization on loss surface contours

Fig. 2.11: Image 6: SGD optimization on saddle point

In Image 5, we see their behaviour on the contours of a loss surface over time. Note that Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast, while Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill. NAG, however, is quickly able to correct its course due to its increased responsiveness by looking ahead and heads to the minimum.

Image 6 shows the behaviour of the algorithms at a saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD as we mentioned before. Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the two latter eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope.

As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.

[15] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.
[32] http://cs231n.github.io/neural-networks-3/

Which optimizer to choose?

So, which optimizer should you now use? If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you won't need to tune the learning rate but likely achieve the best results with the default value.

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al.[15] show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

2.4.3 Parallelizing and distributing SGD

Given the ubiquity of large-scale data solutions and the availability of low-commodity clusters, distributing SGD to speed it up further is an obvious choice.

SGD by itself is inherently sequential: Step-by-step, we progress further towards the minimum. Running it provides good convergence but can be slow particularly on large datasets. In contrast, running SGD asynchronously is faster, but suboptimal communication between workers can lead to poor convergence. Additionally, we can also parallelize SGD on one machine without the need for a large computing cluster. The following are algorithms and architectures that have been proposed to optimize parallelized and distributed SGD.

Hogwild!

Niu et al.[23] introduce an update scheme called Hogwild! that allows performing SGD updates in parallel on CPUs. Processors are allowed to access shared memory without locking the parameters. This only works if the input data is sparse, as each update will only modify a fraction of all parameters. They show that in this case, the update scheme achieves almost an optimal rate of convergence, as it is unlikely that processors will overwrite useful information.

Downpour SGD

Downpour SGD is an asynchronous variant of SGD that was used by Dean et al.[4] in their DistBelief framework (predecessor to TensorFlow) at Google. It runs multiple replicas of a model in parallel on subsets of the training data. These models send their updates to a parameter server, which is split across many machines. Each machine is responsible for storing and updating a fraction of the model's parameters. However, as replicas don't communicate with each other e.g. by sharing weights or updates, their parameters are continuously at risk of diverging, hindering convergence.

[23] Niu, F., Recht, B., Christopher, R., & Wright, S. J. (2011). Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 1–22.

Delay-tolerant Algorithms for SGD

McMahan and Streeter[12] extend AdaGrad to the parallel setting by developing delay-tolerant algorithms that not only adapt to past gradients, but also to the update delays. This has been shown to work well in practice.

TensorFlow

TensorFlow[13] is Google's recently open-sourced framework for the implementation and deployment of large-scale machine learning models. It is based on their experience with DistBelief and is already used internally to perform computations on a large range of mobile devices as well as on large-scale distributed systems. For distributed execution, a computation graph is split into a subgraph for every device and communication takes place using Send/Receive node pairs. However, the open source version of TensorFlow currently does not support distributed functionality (see here[33]). Update 13.04.16: A distributed version of TensorFlow has been released[34].

Elastic Averaging SGD

Zhang et al.[14] propose Elastic Averaging SGD (EASGD), which links the parameters of the workers of asynchronous SGD with an elastic force, i.e. a center variable stored by the parameter server. This allows the local variables to fluctuate further from the center variable, which in theory allows for more exploration of the parameter space. They show empirically that this increased capacity for exploration leads to improved performance by finding new local optima.

2.4.4 Additional strategies for optimizing SGD

Finally, we introduce additional strategies that can be used alongside any of the previously mentioned algorithms to further improve the performance of SGD. For a great overview of some other common tricks, refer to[22].

Shuffling and Curriculum Learning

Generally, we want to avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm. Consequently, it is often a good idea to shuffle the training data after every epoch.

On the other hand, for some cases where we aim to solve progressively harder problems, supplying the training examples in a meaningful order may actually lead to improved performance and better convergence. The method for establishing this meaningful order is called Curriculum Learning[16].

Zaremba and Sutskever[17] were only able to train LSTMs to evaluate simple programs using Curriculum Learning and show that a combined or mixed strategy is better than the naive one, which sorts examples by increasing difficulty.

[12] Mcmahan, H. B., & Streeter, M. (2014). Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. Advances in Neural Information Processing Systems (Proceedings of NIPS), 1–9. Retrieved from http://papers.nips.cc/paper/5242-delay- tolerant-algorithms-for-asynchronous-distributed-online-learning.pdf

[13] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2015). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.

[33] https://github.com/tensorflow/tensorflow/issues/23

[34] http://googleresearch.blogspot.ie/2016/04/announcing-tensorflow-08-now-with.html

[14] Zhang, S., Choromanska, A., & LeCun, Y. (2015). Deep learning with Elastic Averaging SGD. Neural Information Processing Systems Conference (NIPS 2015), 1–24. Retrieved from http://arxiv.org/abs/1412.6651

[22] LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998). Efficient BackProp. Neural Networks: Tricks of the Trade, 1524, 9–50. http://doi.org/10.1007/3- 540-49430-8_2

[16] Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. Proceedings of the 26th Annual International Conference on Machine Learning, 41–48. http://doi.org/10.1145/1553374.1553380

[17] Zaremba, W., & Sutskever, I. (2014). Learning to Execute, 1–25. Retrieved from http://arxiv.org/abs/1410.4615

Batch normalization

To facilitate learning, we typically normalize the initial values of our parameters by initializing them with zero mean and unit variance. As training progresses and we update parameters to different extents, we lose this normalization, which slows down training and amplifies changes as the network becomes deeper.

Batch normalization[18] reestablishes these normalizations for every mini-batch and changes are back-propagated through the operation as well. By making normalization part of the model architecture, we are able to use higher learning rates and pay less attention to the initialization parameters. Batch normalization additionally acts as a regularizer, reducing (and sometimes even eliminating) the need for Dropout.

Early Stopping

According to Geoff Hinton: "Early stopping (is) beautiful free lunch" (NIPS 2015 Tutorial slides, slide 63)[35]. You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.

Gradient noise

Neelakantan et al.[21] add noise that follows a Gaussian distribution $N(0, \sigma_t^2)$ to each gradient update:

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2)$$

They anneal the variance according to the following schedule:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

They show that adding this noise makes networks more robust to poor initialization and helps training particularly deep and complex networks. They suspect that the added noise gives the model more chances to escape and find new local minima, which are more frequent for deeper models.

2.4.5 Conclusion

In this blog post, we have initially looked at the three variants of gradient descent, among which mini-batch gradient descent is the most popular. We have then investigated algorithms that are most commonly used for optimizing SGD: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, as well as different algorithms to optimize asynchronous SGD. Finally, we've considered other strategies to improve SGD such as shuffling and curriculum learning, batch normalization, and early stopping.

2.5 Multilayer Perceptron

2.5.1 Sigmoid function

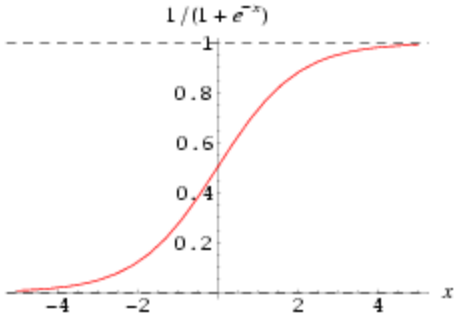
BP algorithm is mainly due to the emergence of Sigmoid function, instead of the previous threshold function to construct neurons.

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

[18] Ioffe, S., & Szegedy, C. (2015). Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv Preprint arXiv:1502.03167v3.

[35] http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf

[21] Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., & Martens, J. (2015). Adding Gradient Noise Improves Learning for Very Deep Networks, 1–11. Retrieved from http://arxiv.org/abs/1511.06807



The Sigmoid function is usually written in the following form:

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

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

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

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.

2.5.2 Back Propagation

Back Propagation (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

$$J(w) = \frac{1}{2} \sum_{j=1}^{s} (t_j - s_j)^2 = \frac{1}{2} \mid t - a \mid^2$$

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

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

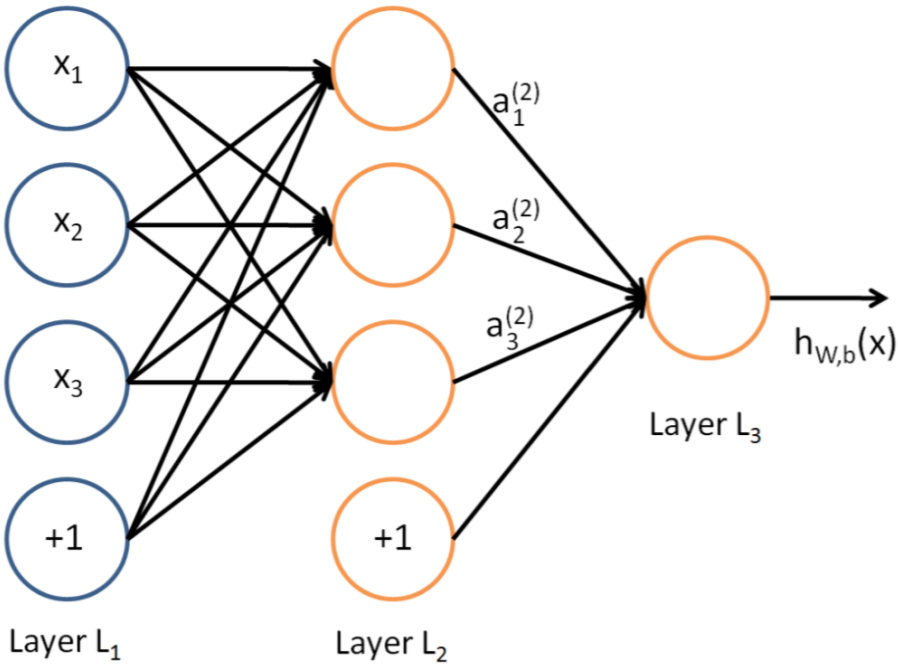
$$\begin{aligned} \bigtriangledown w_j^k &= -\eta \frac{\partial J}{\partial J w_j^k} \\ &= -\eta \frac{\partial J}{\partial n_j^k} \frac{\partial n_j^k}{\partial J w_j^k} \\ &= -\eta \frac{\partial J}{\partial n_j^k} a^{k-1} \\ &= -\eta \delta_j^k a^{k-1} \end{aligned}$$

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 BP 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.

2.5.3 Example

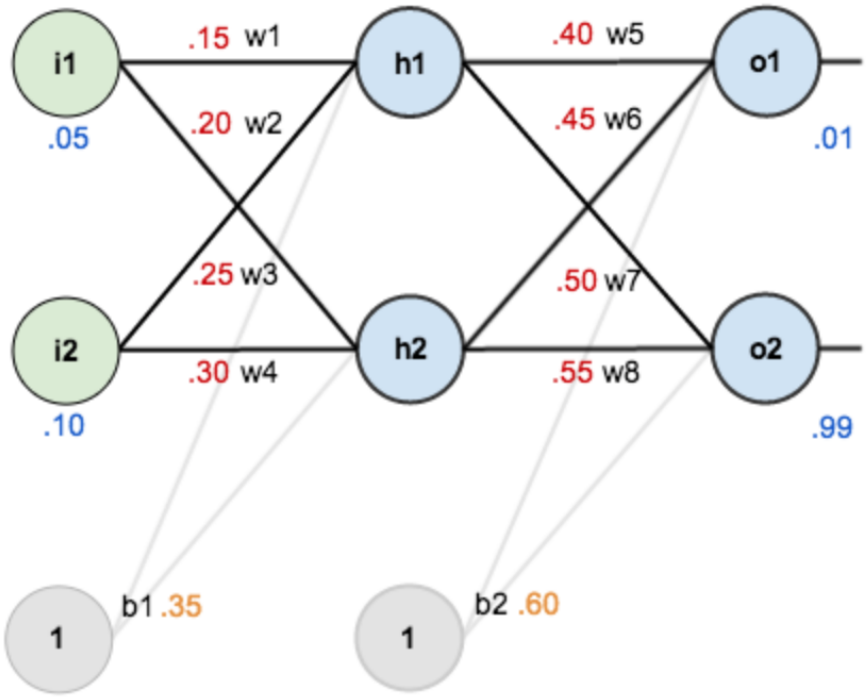
Suppose you have such a network layer:

- The first layer is the input layer, two neurons containing i_1, i_2, b_1 and intercept;
- The second layer is the hidden layer, including two neurons h_1, h_2 and intercept b2;
- The third layer is the output of o_1, o_2 and w_i 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: $i_1 = 0.05, i_2 = 0.10$;
- Output data: $o_1 = 0.01, o_2 = 0.99$;
- Initial weight: $w_1 = 0.15, w_2 = 0.20, w_3 = 0.25, w_4 = 0.30, w_5 = 0.40, w_6 = 0.45, w_7 = 0.50, w_8 = 0.88$;



Objective: to give input data i_1, i_2 (0.05 and 0.10), so that the output is as close as possible to the original output o_1, o_2 (0.01 and 0.99).

Step 1: Forward Propagation

`Input` layer to `Hidden` layer

Calculate the input weighted sum of neurons h_1:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_i * 1$$
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

o_1, the output of neuron h_1: (Activation function sigmoid is required here):

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

Similarly, o_2, the output of neuron h_2 can be calculated:

$$out_{h2} = 0.596884378$$

`Hidden` layer to `Output` layer

The values of o_1 and o_2 in the output layer are calculated:

$$
\begin{aligned}
net_{o1} &= w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1 \\
net_{o1} &= 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967 \\
out_{o1} &= \frac{1}{1 + e^{-net_{o1}}} = \frac{1}{1 + e^{-1.105905967}} = 0.75136507 \\
out_{o2} &= 0.772928465
\end{aligned}
$$

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.

Step 2: Back Propagation

Calculate the total error

Total error (square error):

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

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

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

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

$$E_{o2} = 0.023560026$$

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

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

`Hidden` layer to `Hidden` layer weights update

Take the weight parameter w_5 as an example, if we want to know how much impact the w_5 has on the overall error, we can use the global error to obtain the partial derivative of w_5: (chain rule)

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

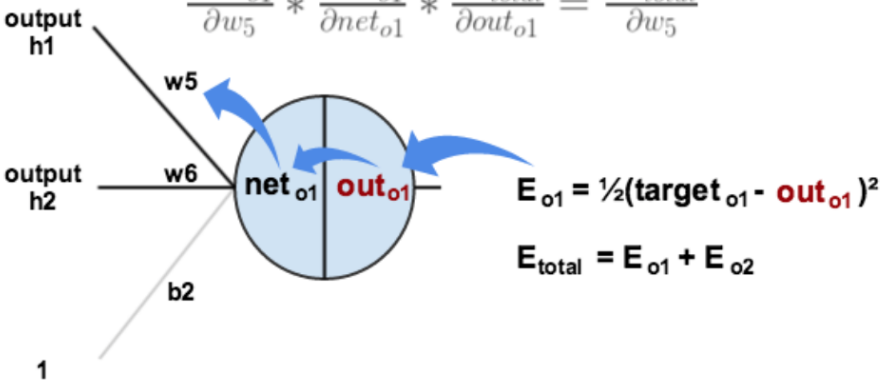
The following figure can be more intuitive to see how the error is spread back:

Now we were calculated for each value:

- Calculate $\frac{\partial E_{total}}{\partial out_{o1}}$.

$$
\begin{aligned}
E_{total} &= \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2 \\
\frac{\partial E_{total}}{\partial out_{o1}} &= 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0 \\
\frac{\partial E_{total}}{\partial out_{o1}} &= -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507
\end{aligned}
$$

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



- Calculate $\frac{\partial out_{o1}}{\partial net_{o1}}$:

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

- Calculate $\frac{\partial net_{o1}}{\partial w_5}$:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$
$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

- Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$
$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.59326992 = 0.082167041$$

In this way, we calculate the overall error E_{total} to the w_5 partial guide. Look at the above formula, we found:

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

In order to express convenience, δ_{o1} is 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 w_5:

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

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 w_5:

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

Among them, η is the learning rate, here we take 0.5. Similarly, update w_6, w_7, w_8:

$$w_6^+ = 0.408666186$$
$$w_7^+ = 0.511301270$$
$$w_8^+ = 0.561370121$$

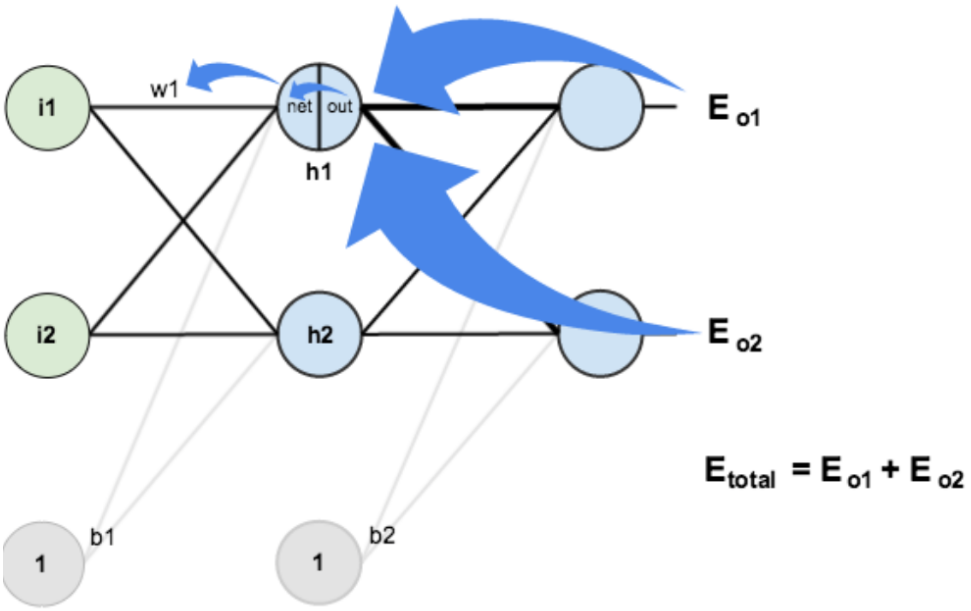
`Hidden` layer to `Input` layer weights update

In fact, with the method above said almost, but there is a need to change, calculate the total error of the above w_5 guide, from out_{o1} —-> net_{o1} —-> w_5, but in the hidden layer between the weight update, out_{h1} —-> net_{h1} —-> w_1 and out_{h1} will accept E_{o1} and E_{o2} error of two places to two, so this place will be calculated.

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

$$\downarrow$$

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



- Calculate $\frac{\partial E_{total}}{\partial out_{h1}}$:

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

- Calculate $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$
$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.74136507 * 0.186815602 = 0.138498562$$
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$
$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$
$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

- Similarly, calculate $\frac{\partial E_{o2}}{\partial out_{h1}} = -.019049119$:
- Therefore,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -.019049119 + 0.036350306$$

- Then, calculate $\frac{\partial out_{h1}}{\partial net_{h1}}$:

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$
$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.241300709$$

- Calculate $\frac{\partial net_{h1}}{\partial w_{h1}}$:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$
$$\frac{\partial net_{h1}}{\partial w_{h1}} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

In order to simplify the formula, σ_{h1} is used to represent the error of the hidden layer unit h_1:

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

We can now update w_1:

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

Repeating this for w_2, w_3, and w_4:

$$w_2^+ = 0.19956143$$
$$w_3^+ = 0.24975114$$
$$w_4^+ = 0.29950229$$

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).

2.5.4 Code

First, import necessary packages:

```
import random
import math
```

Define `network`:

```
class NeuralNetwork:
    LEARNING_RATE = 0.5

    def __init__(self, num_inputs, num_hidden, num_outputs,
                 hidden_layer_weights=None,
                 hidden_layer_bias=None,
                 output_layer_weights=None,
                 output_layer_bias=None):
        self.num_inputs = num_inputs

        self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
        self.output_layer = NeuronLayer(num_outputs, output_layer_bias)

        self.init_weights_from_inputs_to_hidden_layer_neurons(hidden_layer_weights)
        self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(output_
↪layer_weights)

    def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights):
        weight_num = 0
        for h in range(len(self.hidden_layer.neurons)):
            for i in range(self.num_inputs):
                if not hidden_layer_weights:
                    self.hidden_layer.neurons[h].weights.append(random.random())
                else:
                    self.hidden_layer.neurons[h].weights.append(hidden_layer_
↪weights[weight_num])
                weight_num += 1

    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_
↪layer_weights):
        weight_num = 0
        for o in range(len(self.output_layer.neurons)):
            for h in range(len(self.hidden_layer.neurons)):
                if not output_layer_weights:
                    self.output_layer.neurons[o].weights.append(random.random())
                else:
                    self.output_layer.neurons[o].weights.append(output_layer_
↪weights[weight_num])
                weight_num += 1

    def inspect(self):
        print('------')
        print('* Inputs: {}'.format(self.num_inputs))
        print('------')
        print('Hidden Layer')
        self.hidden_layer.inspect()
        print('------')
        print('* Output Layer')
        self.output_layer.inspect()
```

```
        print('------')

    def feed_forward(self, inputs):
        hidden_layer_outputs = self.hidden_layer.feed_forward(inputs)
        return self.output_layer.feed_forward(hidden_layer_outputs)

    # Uses online learning, ie updating the weights after each training case
    def train(self, training_inputs, training_outputs):
        self.feed_forward(training_inputs)

        # 1. Output neuron deltas
        pd_errors_wrt_output_neuron_total_net_input = [0] * len(self.output_layer.
↪neurons)
        for o in range(len(self.output_layer.neurons)):
            # E/z
            pd_errors_wrt_output_neuron_total_net_input[o] = self.output_layer.
↪neurons[
                o].calculate_pd_error_wrt_total_net_input(training_outputs[o])

        # 2. Hidden neuron deltas
        pd_errors_wrt_hidden_neuron_total_net_input = [0] * len(self.hidden_layer.
↪neurons)
        for h in range(len(self.hidden_layer.neurons)):

            # We need to calculate the derivative of the error with respect to the␣
↪output of each hidden layer neuron
            # dE/dy = Σ E/z * z/y = Σ E/z * w
            d_error_wrt_hidden_neuron_output = 0
            for o in range(len(self.output_layer.neurons)):
                d_error_wrt_hidden_neuron_output += pd_errors_wrt_output_neuron_total_
↪net_input[o] * \
                                                    self.output_layer.neurons[o].
↪weights[h]

            # E/z = dE/dy * z/
            pd_errors_wrt_hidden_neuron_total_net_input[h] = d_error_wrt_hidden_
↪neuron_output * \
                                                             self.hidden_layer.
↪neurons[
                                                                 h].calculate_pd_
↪total_net_input_wrt_input()

        # 3. Update output neuron weights
        for o in range(len(self.output_layer.neurons)):
            for w_ho in range(len(self.output_layer.neurons[o].weights)):
                # E/w = E/z * z/w
                pd_error_wrt_weight = pd_errors_wrt_output_neuron_total_net_input[o]␣
↪* self.output_layer.neurons[
                    o].calculate_pd_total_net_input_wrt_weight(w_ho)

                # Δw = α * E/w
                self.output_layer.neurons[o].weights[w_ho] -= self.LEARNING_RATE * pd_
↪error_wrt_weight

        # 4. Update hidden neuron weights
        for h in range(len(self.hidden_layer.neurons)):
            for w_ih in range(len(self.hidden_layer.neurons[h].weights)):
                # E/w = E/z * z/w
```

```
                pd_error_wrt_weight = pd_errors_wrt_hidden_neuron_total_net_input[h]␣
↪* self.hidden_layer.neurons[
                    h].calculate_pd_total_net_input_wrt_weight(w_ih)

                # ∆w = α * E/w
                self.hidden_layer.neurons[h].weights[w_ih] -= self.LEARNING_RATE * pd_
↪error_wrt_weight

    def calculate_total_error(self, training_sets):
        total_error = 0
        for t in range(len(training_sets)):
            training_inputs, training_outputs = training_sets[t]
            self.feed_forward(training_inputs)
            for o in range(len(training_outputs)):
                total_error += self.output_layer.neurons[o].calculate_error(training_
↪outputs[o])
        return total_error
```

Define `layer`:

```
class NeuronLayer:
    def __init__(self, num_neurons, bias):

        # Every neuron in a layer shares the same bias
        self.bias = bias if bias else random.random()

        self.neurons = []
        for i in range(num_neurons):
            self.neurons.append(Neuron(self.bias))

    def inspect(self):
        print('Neurons:', len(self.neurons))
        for n in range(len(self.neurons)):
            print(' Neuron', n)
            for w in range(len(self.neurons[n].weights)):
                print('  Weight:', self.neurons[n].weights[w])
            print('  Bias:', self.bias)

    def feed_forward(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

    def get_outputs(self):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.output)
        return outputs
```

Define `neuron`: .. literalinclude:: mlp_bp.py

start-after neuron-start

end-before neuron-end

Put all together, and run example:

```
nn = NeuralNetwork(2, 2, 2,
                   hidden_layer_weights=[0.15, 0.2, 0.25, 0.3],
                   hidden_layer_bias=0.35,
                   output_layer_weights=[0.4, 0.45, 0.5, 0.55],
                   output_layer_bias=0.6)
for i in range(10000):
    nn.train([0.05, 0.1], [0.01, 0.99])
    print(i, round(nn.calculate_total_error([[[0.05, 0.1], [0.01, 0.99]]]), 9))

    # XOR example:

    # training_sets = [
    #     [[0, 0], [0]],
    #     [[0, 1], [1]],
    #     [[1, 0], [1]],
    #     [[1, 1], [0]]
    # ]

    # nn = NeuralNetwork(len(training_sets[0][0]), 5, len(training_sets[0][1]))
    # for i in range(10000):
    #     training_inputs, training_outputs = random.choice(training_sets)
    #     nn.train(training_inputs, training_outputs)
    #     print(i, nn.calculate_total_error(training_sets))
```

Please Enjoy!

2.6 Convolution Neural Networks

2.6.1 Introduction

Convolutional Neural Networks (CNN) are now a standard way of image classification – there are publicly accessible deep learning frameworks, trained models and services. It's more time consuming to install stuff like caffe[1] than to perform state-of-the-art object classification or detection. We also have many methods of getting knowledge - there is a large number of deep learning courses[2] /MOOCs[3], free e-books[4] or even direct ways of accessing to the strongest Deep/Machine Learning minds such as Yoshua Bengio[5], Andrew NG[6] or Yann Lecun[7] by Quora, Facebook or G+.

Nevertheless, when I wanted to get deeper insight in CNN, I could not find a "CNN backpropagation for dummies". Notoriously I met with statements like: "If you understand backpropagation in standard neural networks, there should not be a problem with understanding it in CNN" or "All things are nearly the same, except matrix multiplications are replaced by convolutions". And of course I saw tons of ready equations.

It was a little consoling, when I found out that I am not alone, for example: `Hello, when computing the`

[1] http://caffe.berkeleyvision.org/
[2] http://cs224d.stanford.edu/
[3] https://www.udacity.com/course/deep-learning–ud730
[4] http://www.deeplearningbook.org/
[5] https://plus.google.com/+YoshuaBengio/posts
[6] https://www.quora.com/session/Andrew-Ng/1
[7] https://www.facebook.com/yann.lecun?fref=ts

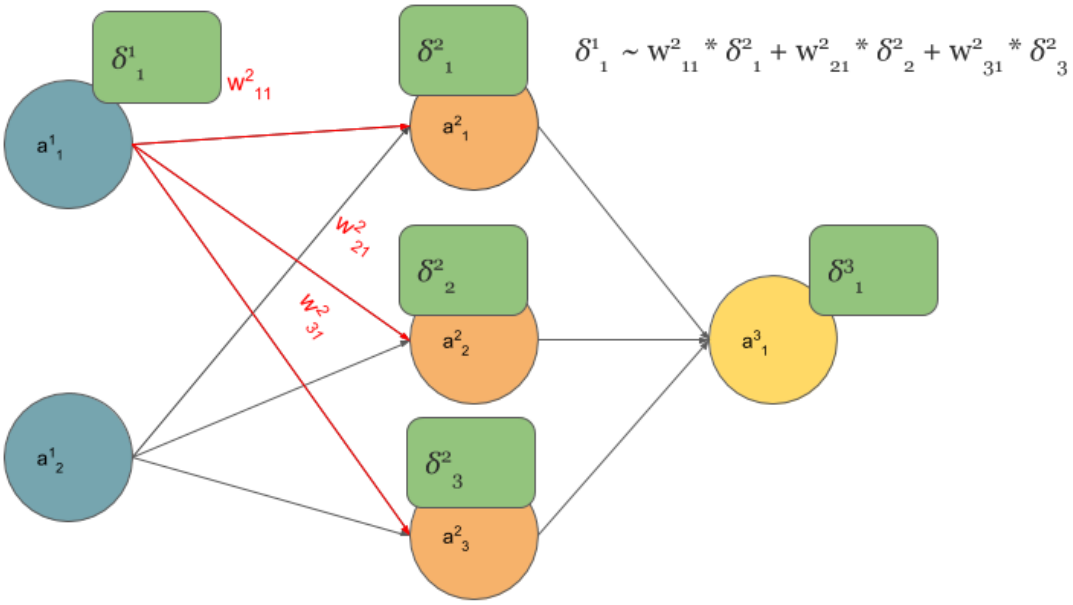
`gradients CNN, the weights need to be rotated, Why ?`[8]

$$\delta_j^l = f'(u_j^l) \odot conv2(\delta_j^{l+1}, rot180(k_j^{l+1}),' full')$$

The answer on above question, that concerns the need of rotation on weights in gradient computing, will be a result of this long post.

2.6.2 Back Propagation

We start from multilayer perceptron and counting delta errors on fingers:



We see on above picture that δ_1^1 is proportional to deltas from next layer that are scaled by weights.

But how do we connect concept of MLP with Convolutional Neural Network? Let's play with MLP:

If you are not sure that after connections cutting and weights sharing we get one layer Convolutional Neural Network, I hope that below picture will convince you:

The idea behind this figure is to show, that such neural network configuration is identical with a 2D convolution operation and weights are just filters (also called kernels, convolution matrices, or masks).

Now we can come back to gradient computing by counting on fingers, but from now we will be only focused on CNN. Let's begin:

No magic here, we have just summed in "blue layer" scaled by weights gradients from "orange" layer. Same process as in MLP's backpropagation. However, in the standard approach we talk about dot products and here we have ... yup, again convolution:

Yeah, it is a bit different convolution than in previous (forward) case. There we did so called valid convolution, while here we do a full convolution (more about nomenclature here[9]). What is more, we rotate our kernel by 180 degrees. But still, we are talking about convolution!

Now, I have some good news and some bad news:

[8] https://plus.google.com/115541909707081118542/posts/P8bZBNpg84Z
[9] http://www.johnloomis.org/ece563/notes/filter/conv/convolution.html

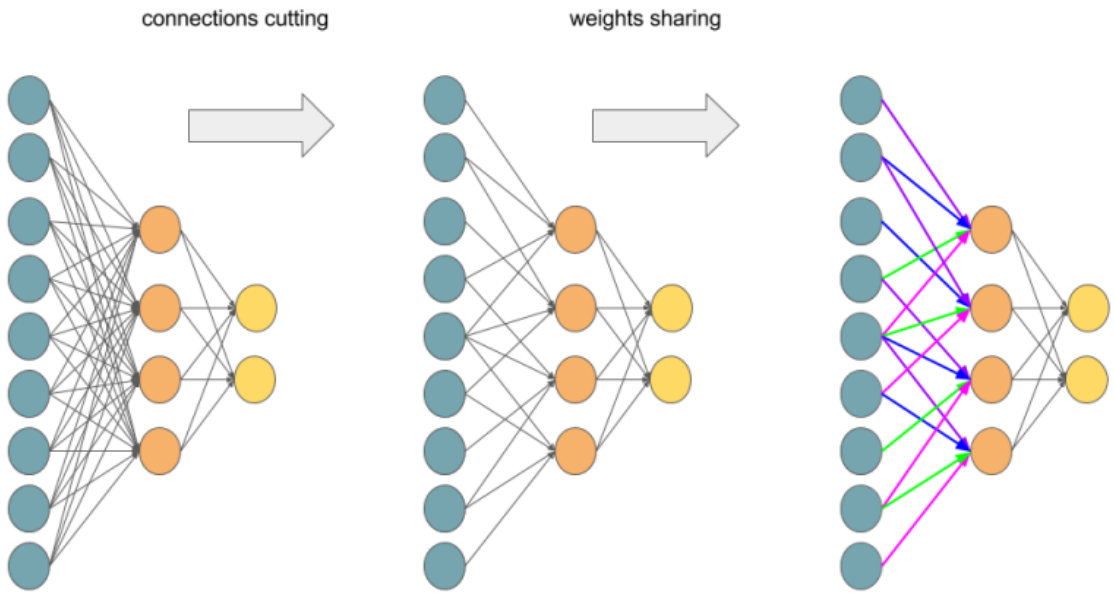


Fig. 2.12: Transforming Multilayer Perceptron to Convolutional Neural Network.

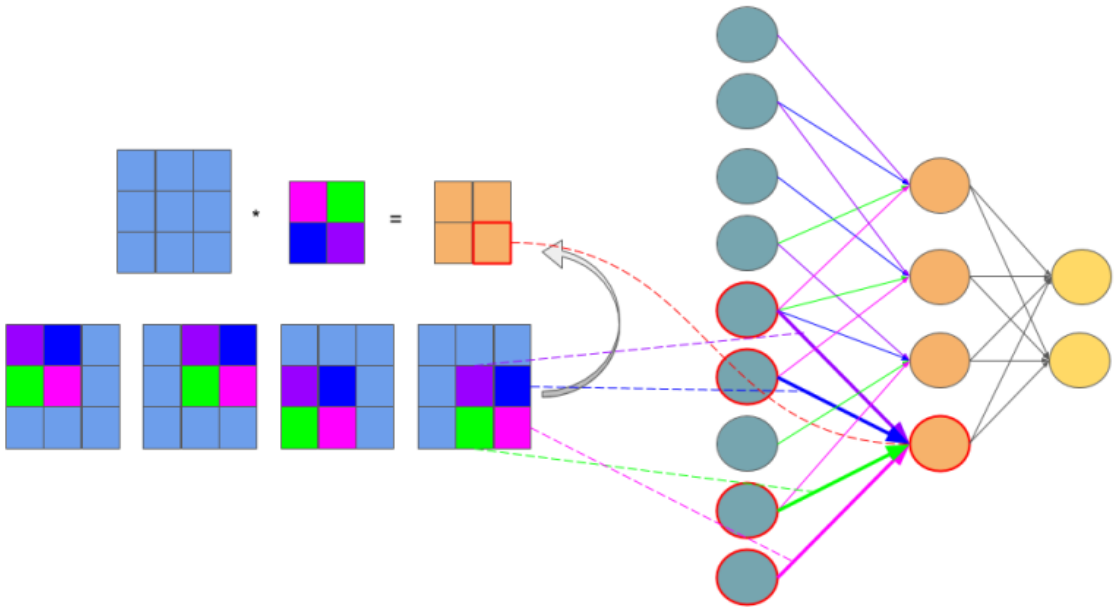


Fig. 2.13: Feedforward in CNN is identical with convolution operation.

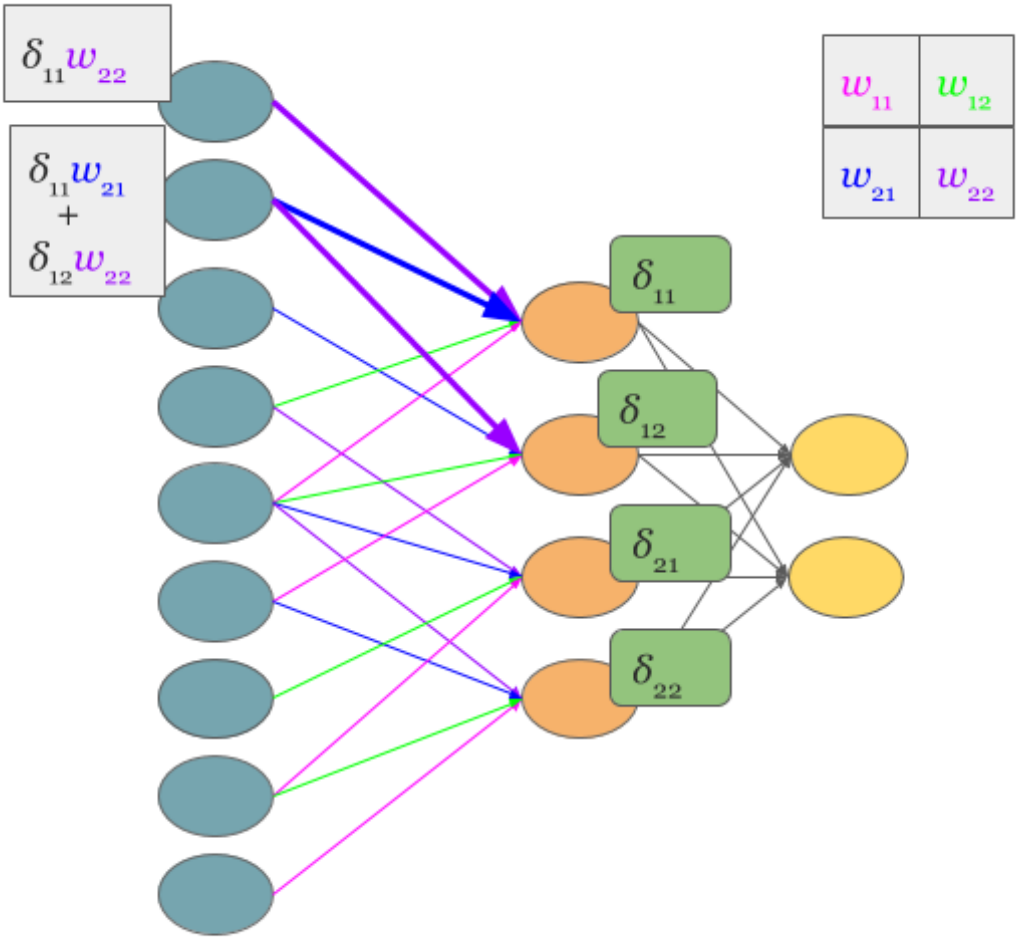
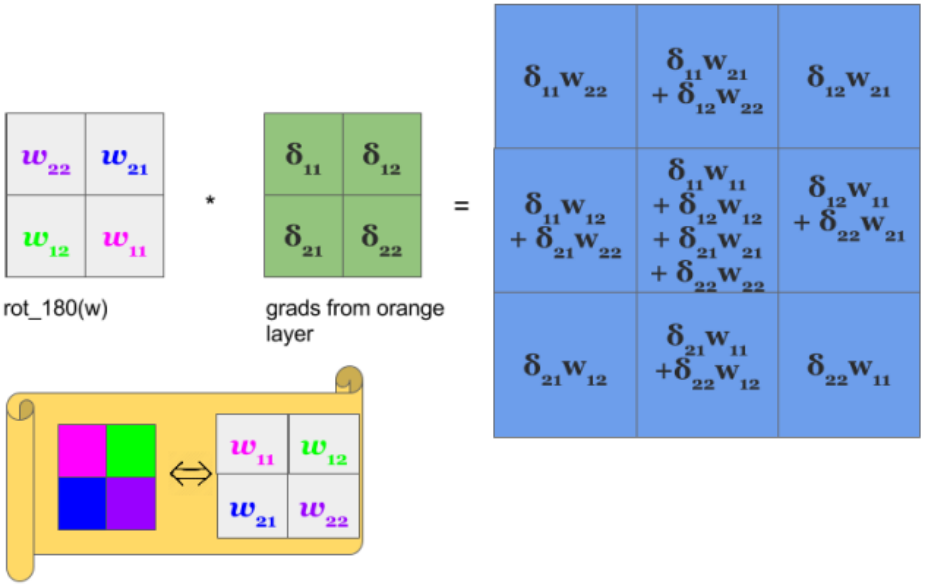
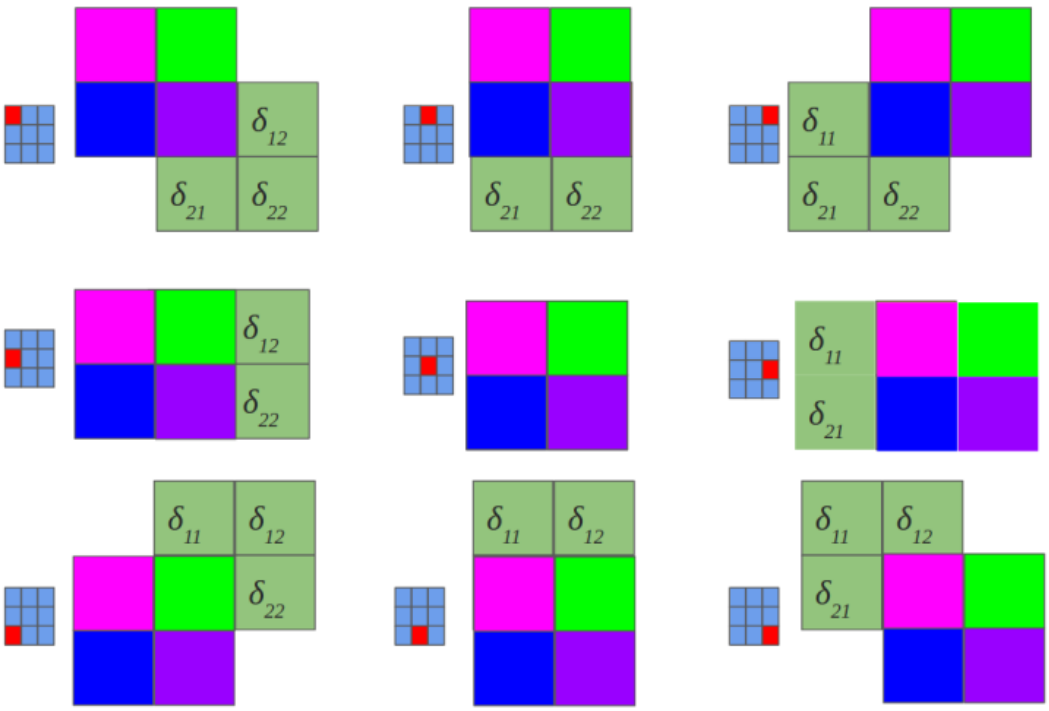


Fig. 2.14: Backpropagation also results with convolution.


rot_180(w)
grads from orange layer



- you see (BTW, sorry for pictures aesthetics), that matrix dot products are replaced by convolution operations both in feed forward and backpropagation.
- you know that seeing something and understanding something ... yup, we are going now to get our hands dirty and prove above statement before getting next, I recommend to read, mentioned already in the disclaimer, chapter 2[10] of M. Nielsen book. I tried to make all quantities to be consistent with work of Michael.

In the standard MLP, we can define an error of neuron j as:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

where z_j^l is just:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

and for clarity, $a_j^l = \sigma(z_j^l)$, where σ is an activation function such as sigmoid, hyperbolic tangent or relu[11].

But here, we do not have MLP but CNN and matrix multiplications are replaced by convolutions as we discussed before. So instead of z_j we do have a $z_{x,y}$:

$$z_{x,y}^{l+1} = w^{l+1} * \sigma(z_{x,y}^l) + b_{x,y}^{l+1} = \sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x-a,y-b}^l) + b_{x,y}^{l+1}$$

Above equation is just a convolution operation during feedforward phase illustrated in the above picture titled 'Feedforward in CNN is identical with convolution operation'[12]

Now we can get to the point and answer the question `Hello, when computing the gradients CNN, the weights need to be rotated, Why ?`[8]

We start from statement:

$$\delta_{x,y}^l = \frac{\partial C}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z_{x',y'}^{l+1}} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l}$$

We know that $z_{x,y}^l$ is in relation to $z_{x',y'}^{l+1}$ which is indirectly showed in the above picture titled 'Backpropagation also results with convolution'. So sums are the result of chain rule. Let's move on:

$$\begin{aligned}\frac{\partial C}{\partial z_{x,y}^l} &= \sum_{x'} \sum_{y'} \frac{\partial C}{\partial z_{x',y'}^{l+1}} \frac{\partial z_{x',y'}^{l+1}}{\partial z_{x,y}^l} \\ &= \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} \frac{\partial (\sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x'-a,y'-b}^l) + b_{x',y'}^{l+1})}{\partial z_{x,y}^l}\end{aligned}$$

First term is replaced by definition of error, while second has become large because we put it here expression on $z_{x',y'}^{l+1}$. However, we do not have to fear of this big monster – all components of sums equal 0, except these ones that are indexed: $x = x' - a$ and $y = y' - b$. So:

$$\sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} \frac{\partial (\sum_a \sum_b w_{a,b}^{l+1} \sigma(z_{x'-a,y'-b}^l) + b_{x',y'}^{l+1})}{\partial z_{x,y}^l} = \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{a,b}^{l+1} \sigma'(z_{x,y}^l)$$

If "math;'x=x'-a' and $y = y' - b$ then it is obvious that $a = x' - x$ and $b = y' - y$ so we can reformulate above equation to:

$$\sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{a,b}^{l+1} \sigma'(z_{x,y}^l) = \sum_{x'} \sum_{y'} \delta_{x',y'}^{l+1} w_{x'-x,y'-y}^{l+1} \sigma'(z_{x,y}^l)$$

[10] http://neuralnetworksanddeeplearning.com/chap2.html
[11] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[12] https://grzegorzgwardys.wordpress.com/2016/04/22/8/#unique-identifier

OK, our last equation is just ...

$$\sum_{x'}\sum_{y'}\delta^{l+1}_{x',y'}w^{l+1}_{x'-x,y'-y}\sigma'(z^l_{x,y}) = \delta^{l+1} * w^{l+1}_{-x,-y}\sigma'(z^l_{x,y})$$

Where is the rotation of weights? Actually $ROT180(w^{l+1}_{x,y}) = w^{l+1}_{-x,-y}$.

So the answer on question `Hello, when computing the gradients CNN, the weights need to be rotated, Why ?`[8] is simple: the rotation of the weights just results from derivation of delta error in Convolution Neural Network.

OK, we are really close to the end. One more ingredient of backpropagation algorithm is update of weights $\frac{\partial C}{\partial w^l_{a,b}}$:

$$\begin{aligned}
\frac{\partial C}{\partial w^l_{a,b}} &= \sum_x\sum_y \frac{\partial C}{\partial z^l_{x,y}}\frac{\partial z^l_{x,y}}{\partial w^l_{a,b}} \\
&= \sum_x\sum_y \delta^l_{x,y}\frac{\partial(\sum_{a'}\sum_{b'} w^l_{a',b'}\sigma(z^l_{x-a',y-b'}) + b^l_{x,y})}{\partial w^l_{a,b}} \\
&= \sum_x\sum_y \delta^l_{x,y}\sigma(z^{l-1}_{x-a,y-b}) \\
&= \delta^l_{a,b} * \sigma(z^{l-1}_{-a,-b}) \\
&= \delta^l_{a,b} * \sigma(ROT180(z^{l-1}_{a,b}))
\end{aligned}$$

So paraphrasing the backpropagation algorithm[13] for CNN:

1. Input x: set the corresponding activation a^1 for the input layer.
2. Feedforward: for each $l = 2, 3, \cdots, L$, compute $z^l_{x,y} = w^l * \sigma(z^{l-1}_{x,y}) + b^l_{x,y}$ and $a^l_{x,y} = \sigma(z^l_{x,y})$
3. Output error δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. Backpropagate the error: For each $l = L-1, L-2, \cdots, 2$, compute $\delta^l_{x,y} = \delta^{l+1} * ROT180(w^{l+1}_{x,y})\sigma'(z^l_{x,y})$
5. Output: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{a,b}} = \delta^l_{a,b} * \sigma(ROT180(z^{l-1}_{a,b}))$

2.6.3 Visualizing Features

It's been shown many times that convolutional neural nets are very good at recognizing patterns in order to classify images. But what patterns are they actually looking for?

I attempted to recreate the techniques described in[14] to project features in the convnet back to pixel space.

In order to do this, we first need to define and train a convolutional network. Due to lack of training power, I couldn't train on ImageNet and had to use CIFAR-10, a dataset of $32x32$ images in 10 classes. The network structure was pretty standard: two convolutional layers, each with $2x2$ max pooling and a reLu gate, followed by a fully-connected layer and a softmax classifier.

We're only trying to visualize the features in the convolutional layers, so we can effectively ignore the fully-connected and softmax layers.

Features in a convolutional network are simply numbers that represent how present a certain pattern is. The intuition behind displaying these features is pretty simple: we input one image, and retrieve the matrix of features. We set every feature to 0 except one, and pass it backwards through the network until reaching the pixel layer. The challenge here lies in how to effectively pass data backwards through a convolutional network.

[13] http://neuralnetworksanddeeplearning.com/chap2.html#the_backpropagation_algorithm

[14] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." European conference on computer vision. Springer International Publishing, 2014.

We can approach this problem step-by-step. There are three main portions to a convolutional layer. The actual convolution, some max-pooling, and a nonlinearity (in our case, a rectified linear unit). If we can figure out how to calculate the inputs to these units given their outputs, we can pass any feature back to the pixel input.

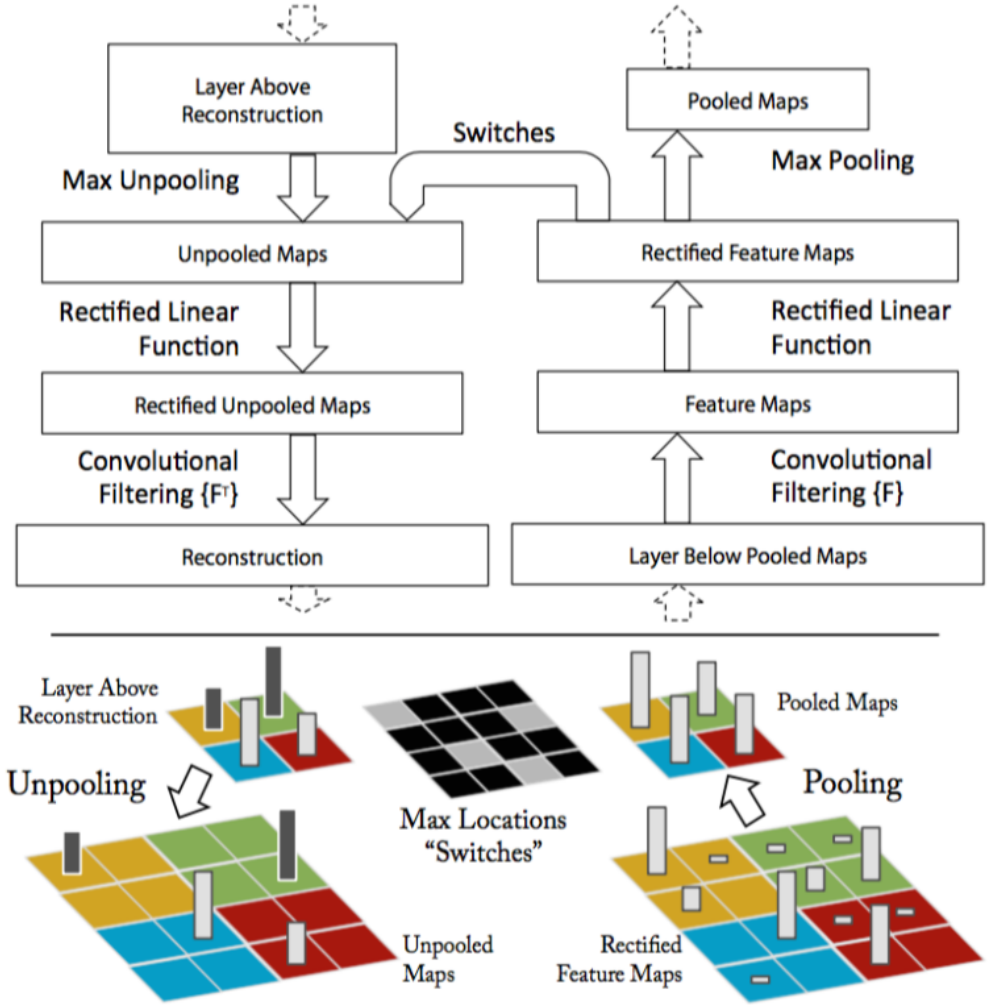


Fig. 2.15: image from[14].

Here, the paper introduces a structure called a deconvolutional layer. However, in practice, this is simply a regular convolutional layer with its filters transposed. By applying these transposed filters to the output of a convolutional layer, the input can be retrieved.

A max-pool gate cannot be reversed on its own, as data about the non-maximum features is lost. The paper describes a method in which the positions of each maximum is recorded and saved during forward propagation, and when features are passed backwards, they are placed where the maximums had originated from. In my recreation, I took an even simpler route and just set the whole $2x2$ square equal to the maximum activation.

Finally, the rectified linear unit. It's the easiest one to reverse, we just need to pass the data through a reLu again when moving backwards.

To test these techniques out, I trained a standard convolutional network on CIFAR-10. First, I passed one of the training images, a dog, through the network and recorded the various features.

Fig. 2.16: our favorite dog.

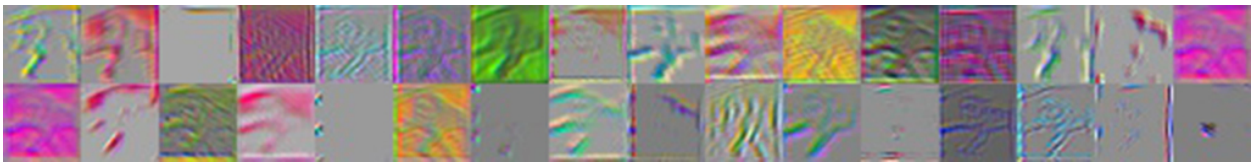


Fig. 2.17: first convolutional layer (features 1-32).

As you can see, there are quite a variety of patterns the network is looking for. You can see evidence of the original dog picture in these feature activations, most prominently the arms.

Now, let's see how these features change when different images are passed through.

Fig. 2.18: first convolutional layer (feature #7).

This image shows all the different pixel representations of the activations of feature #7, when a variety of images are used. It's clear that this feature activates when green is present. You can really see the original picture in this feature, since it probably just captures the overall color green rather than some specific pattern.

Finally, to gain some intuition of how images activated each feature, I passed in a whole batch of images and saved the maximum activations.

Which features were activated by which images? There's some interesting stuff going on here. Some of the features

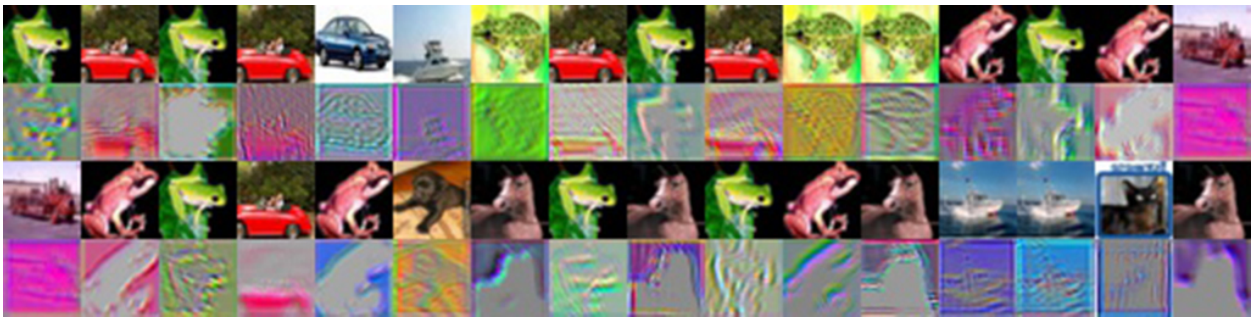


Fig. 2.19: maximum activations for 32 features.

are activated simply by the presence of a certain color. The green frog and red car probably contained the most of their respective colors in the batch of images.

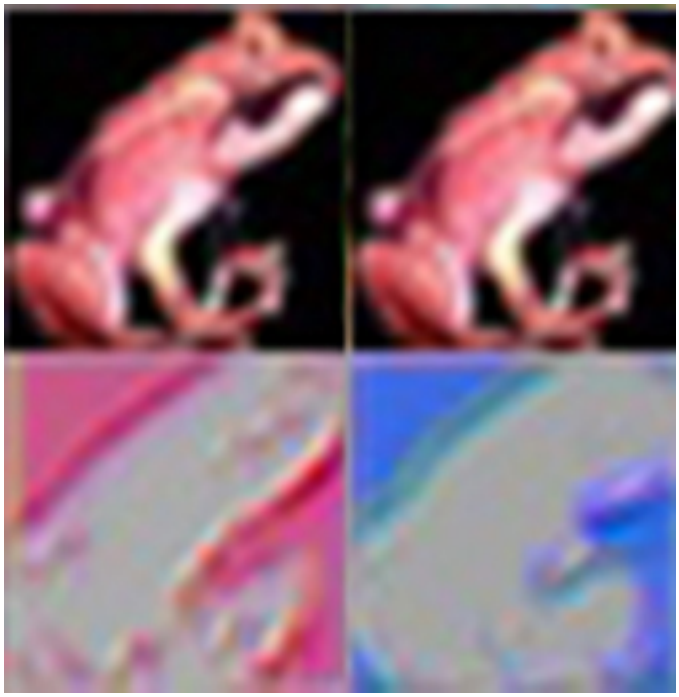


Fig. 2.20: two activations from the above image.

However, here are two features which are activated the most by a red frog image. The feature activations show an outline, but one is in red and the other is in blue. Most likely, this feature isn't getting activated by the frog itself, but by the black background. Visualizing the features of a convolutional network allows us to see such details.

So, what happens if we go farther, and look at the second convolutional layer?

I took the feature activations for the dog again, this time on the second convolutional layer. Already some differences can be spotted. The presence of the original image here is much harder to see.

It's a good sign that all the features are activated in different places. Ideally, we want features to have minimal correlation with one another.

Finally, let's examine how a second layer feature activates when various images are passed in.

For the majority of these images, feature #9 activated at dark locations of the original image. However, there are still

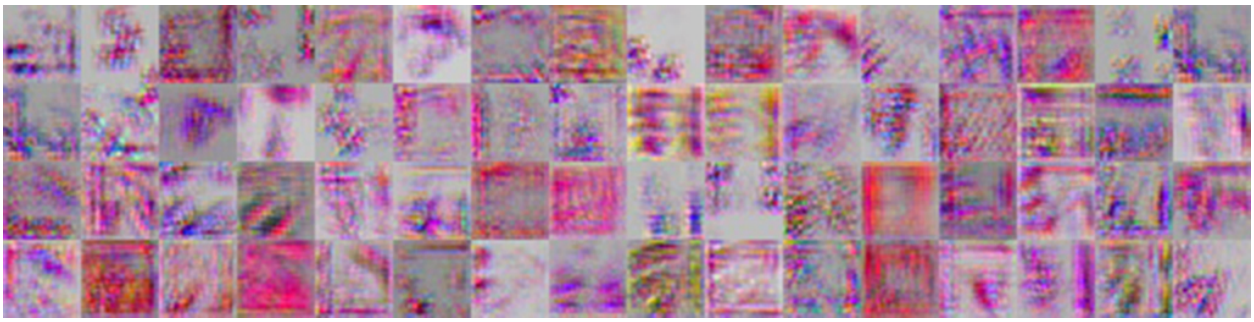


Fig. 2.21: second convolutional layer (64 features).

Fig. 2.22: second convolutional layer (feature #9).

outliers to this, so there is probably more to this feature than that.

For most features, it's a lot harder to tell what part of the image activated it, since second layer features are made of any linear combination of first layer features. I'm sure that if the network was trained on a higher resolution image set, these features would become more apparent.

2.6.4 Code

```
try:
    import tensorflow as tf
    import numpy as np
    import pickle
    from tensorflow.python.platform import gfile
    from random import randint
    import os
    from scipy.misc import imsave
    from matplotlib import pyplot as plt
except ImportError:
    raise ValueError("Please install tensorflow and matplotlib.")


def unpickle(file):
    fo = open(file, 'rb')
    dict = pickle.load(fo)
    fo.close()
    return dict


def initWeight(shape):
    weights = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(weights)


# start with 0.1 so reLu isnt always 0
def initBias(shape):
    bias = tf.constant(0.1, shape=shape)
    return tf.Variable(bias)


# the convolution with padding of 1 on each side, and moves by 1.
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding="SAME")


# max pooling basically shrinks it by 2x, taking the highest value on each feature.
def maxPool2d(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")


batchsize = 50
imagesize = 32
colors = 3

sess = tf.InteractiveSession()

img = tf.placeholder("float", shape=[None, imagesize, imagesize, colors])
lbl = tf.placeholder("float", shape=[None, 10])
```

```
# for each 5x5 area, check for 32 features over 3 color channels
wConv1 = initWeight([5, 5, colors, 32])
bConv1 = initBias([32])
# move the conv filter over the picture
conv1 = conv2d(img, wConv1)
# adds bias
bias1 = conv1 + bConv1
# relu = max(0,x), adds nonlinearality
relu1 = tf.nn.relu(bias1)
# maxpool to 16x16
pool1 = maxPool2d(relu1)
# second conv layer, takes a 16x16 with 32 layers, turns to 8x8 with 64 layers
wConv2 = initWeight([5, 5, 32, 64])
bConv2 = initBias([64])
conv2 = conv2d(pool1, wConv2)
bias2 = conv2 + bConv2
relu2 = tf.nn.relu(bias2)
pool2 = maxPool2d(relu2)
# fully-connected is just a regular neural net: 8*8*64 for each training data
wFc1 = initWeight([(imagesize / 4) * (imagesize / 4) * 64, 1024])
bFc1 = initBias([1024])
# reduce dimensions to flatten
pool2flat = tf.reshape(pool2, [-1, (imagesize / 4) * (imagesize / 4) * 64])
# 128 training set by 2304 data points
fc1 = tf.matmul(pool2flat, wFc1) + bFc1
relu3 = tf.nn.relu(fc1)
# dropout removes duplicate weights
keepProb = tf.placeholder("float")
drop = tf.nn.dropout(relu3, keepProb)
wFc2 = initWeight([1024, 10])
bFc2 = initWeight([10])
# softmax converts individual probabilities to percentages
guesses = tf.nn.softmax(tf.matmul(drop, wFc2) + bFc2)
# how wrong it is
cross_entropy = -tf.reduce_sum(lbl * tf.log(guesses + 1e-9))
# theres a lot of tensorflow optimizers such as gradient descent
# adam is one of them
optimizer = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
# array of bools, checking if each guess was correct
correct_prediction = tf.equal(tf.argmax(guesses, 1), tf.argmax(lbl, 1))
# represent the correctness as a float [1,1,0,1] -> 0.75
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

sess.run(tf.initialize_all_variables())

batch = unpickle("cifar-10-batches-py/data_batch_1")

validationData = batch["data"][555:batchsize + 555]
validationRawLabel = batch["labels"][555:batchsize + 555]
validationLabel = np.zeros((batchsize, 10))
validationLabel[np.arange(batchsize), validationRawLabel] = 1
validationData = validationData / 255.0
validationData = np.reshape(validationData, [-1, 3, 32, 32])
validationData = np.swapaxes(validationData, 1, 3)

saver = tf.train.Saver()
saver.restore(sess, tf.train.latest_checkpoint(os.getcwd() + "/training/"))
```

```
# train for 20000
# print mnistbatch[0].shape
def train():
    for i in range(20000):
        randomint = randint(0, 10000 - batchsize - 1)
        trainingData = batch["data"][randomint:batchsize + randomint]
        rawlabel = batch["labels"][randomint:batchsize + randomint]
        trainingLabel = np.zeros((batchsize, 10))
        trainingLabel[np.arange(batchsize), rawlabel] = 1
        trainingData = trainingData / 255.0
        trainingData = np.reshape(trainingData, [-1, 3, 32, 32])
        trainingData = np.swapaxes(trainingData, 1, 3)

        if i % 10 == 0:
            train_accuracy = accuracy.eval(feed_dict={
                img: validationData, lbl: validationLabel, keepProb: 1.0})
            print("step %d, training accuracy %g" % (i, train_accuracy))

            if i % 50 == 0:
                saver.save(sess, os.getcwd() + "/training/train", global_step=i)

        optimizer.run(feed_dict={img: trainingData, lbl: trainingLabel, keepProb: 0.5}
↪)
        print(i)


def unpool(value, name='unpool'):
    """N-dimensional version of the unpooling operation from
    https://www.robots.ox.ac.uk/~vgg/rg/papers/Dosovitskiy_Learning_to_Generate_2015_
↪CVPR_paper.pdf
    :param value: A Tensor of shape [b, d0, d1, ..., dn, ch]
    :return: A Tensor of shape [b, 2*d0, 2*d1, ..., 2*dn, ch]
    """
    with tf.name_scope(name) as scope:
        sh = value.get_shape().as_list()
        dim = len(sh[1:-1])
        out = (tf.reshape(value, [-1] + sh[-dim:]))
        for i in range(dim, 0, -1):
            out = tf.concat(i, [out, out])
        out_size = [-1] + [s * 2 for s in sh[1:-1]] + [sh[-1]]
        out = tf.reshape(out, out_size, name=scope)
    return out


def display():
    print("displaying")

    batchsizeFeatures = 50
    imageIndex = 56

    inputImage = batch["data"][imageIndex:imageIndex + batchsizeFeatures]
    inputImage = inputImage / 255.0
    inputImage = np.reshape(inputImage, [-1, 3, 32, 32])
    inputImage = np.swapaxes(inputImage, 1, 3)

    inputLabel = np.zeros((batchsize, 10))
    inputLabel[np.arange(1), batch["labels"][imageIndex:imageIndex +␣
↪batchsizeFeatures]] = 1
```

```
    # inputLabel = batch["labels"][54]

    # prints a given image

    # saves pixel-representations of features from Conv layer 1
    featuresReLu1 = tf.placeholder("float", [None, 32, 32, 32])
    unReLu = tf.nn.relu(featuresReLu1)
    unBias = unReLu
    unConv = tf.nn.conv2d_transpose(unBias, wConv1, output_shape=[batchsizeFeatures,␣
↪imagesize, imagesize, colors],
                                    strides=[1, 1, 1, 1], padding="SAME")
    activations1 = relu1.eval(feed_dict={img: inputImage, lbl: inputLabel, keepProb:␣
↪1.0})
    print(np.shape(activations1))

    # display features
    for i in range(32):
        isolated = activations1.copy()
        isolated[:, :, :, :i] = 0
        isolated[:, :, :, i + 1:] = 0
        print(np.shape(isolated))
        totals = np.sum(isolated, axis=(1, 2, 3))
        best = np.argmin(totals, axis=0)
        print(best)
        pixelactive = unConv.eval(feed_dict={featuresReLu1: isolated})
        # totals = np.sum(pixelactive,axis=(1,2,3))
        # best = np.argmax(totals,axis=0)
        # best = 0
        saveImage(pixelactive[best], "activ" + str(i) + ".png")
        saveImage(inputImage[best], "activ" + str(i) + "-base.png")

    # display same feature for many images
    # for i in xrange(batchsizeFeatures):
    #     isolated = activations1.copy()
    #     isolated[:,:,:,:6] = 0
    #     isolated[:,:,:,7:] = 0
    #     pixelactive = unConv.eval(feed_dict={featuresReLu1: isolated})
    #     totals = np.sum(pixelactive,axis=(1,2,3))
    #     best = np.argmax(totals,axis=0)
    #     saveImage(pixelactive[i],"activ"+str(i)+".png")
    #     saveImage(inputImage[i],"activ"+str(i)+"-base.png")



    # saves pixel-representations of features from Conv layer 2
    featuresReLu2 = tf.placeholder("float", [None, 16, 16, 64])
    unReLu2 = tf.nn.relu(featuresReLu2)
    unBias2 = unReLu2
    unConv2 = tf.nn.conv2d_transpose(unBias2, wConv2,
                                     output_shape=[batchsizeFeatures, imagesize / 2,␣
↪imagesize / 2, 32],
                                     strides=[1, 1, 1, 1], padding="SAME")
    unPool = unpool(unConv2)
    unReLu = tf.nn.relu(unPool)
    unBias = unReLu
    unConv = tf.nn.conv2d_transpose(unBias, wConv1, output_shape=[batchsizeFeatures,␣
↪imagesize, imagesize, colors],
                                    strides=[1, 1, 1, 1], padding="SAME")
```

```
    activations1 = relu2.eval(feed_dict={img: inputImage, lbl: inputLabel, keepProb:␣
↪1.0})
    print(np.shape(activations1))

    # display features
    # for i in xrange(64):
    #     isolated = activations1.copy()
    #     isolated[:,:,:,:i] = 0
    #     isolated[:,:,:,i+1:] = 0
    #     pixelactive = unConv.eval(feed_dict={featuresReLu2: isolated})
    #     # totals = np.sum(pixelactive,axis=(1,2,3))
    #     # best = np.argmax(totals,axis=0)
    #     best = 0
    #     saveImage(pixelactive[best],"activ"+str(i)+".png")
    #     saveImage(inputImage[best],"activ"+str(i)+"-base.png")


    # display same feature for many images
    # for i in xrange(batchsizeFeatures):
    #     isolated = activations1.copy()
    #     isolated[:,:,:,:8] = 0
    #     isolated[:,:,:,9:] = 0
    #     pixelactive = unConv.eval(feed_dict={featuresReLu2: isolated})
    #     totals = np.sum(pixelactive,axis=(1,2,3))
    #     # best = np.argmax(totals,axis=0)
    #     # best = 0
    #     saveImage(pixelactive[i],"activ"+str(i)+".png")
    #     saveImage(inputImage[i],"activ"+str(i)+"-base.png")


def saveImage(inputImage, name):
    # red = inputImage[:1024]
    # green = inputImage[1024:2048]
    # blue = inputImage[2048:]
    # formatted = np.zeros([3,32,32])
    # formatted[0] = np.reshape(red,[32,32])
    # formatted[1] = np.reshape(green,[32,32])
    # formatted[2] = np.reshape(blue,[32,32])
    # final = np.swapaxes(formatted,0,2)/255
    final = inputImage
    final = np.rot90(np.rot90(np.rot90(final)))
    imsave(name, final)


def main(argv=None):
    display()
    # train()


if __name__ == '__main__':
    tf.app.run()
```

2.7 Recurrent Neural Networks

Last post, we talked about the traditional feed-forward neural net and concepts that form the basis of deep learning. These ideas are extremely powerful! We saw how feed-forward convolutional neural networks have set records on many difficult tasks including handwritten digit recognition and object classification. And even today, feed-forward neural networks consistently outperform virtually all other approaches to solving classification tasks.

And yet, despite their well celebrated successes, most experts would agree that feed-forward neural nets are still rather limited in what they can achieve. Why? Because the task of "classification" is only one small component of the incredible computational power of the human brain. We're wired not only to recognize individual instances but to also analyze entire sequences of inputs. These sequences are ultra rich in information, have complex time dependencies, and can be of arbitrary length. For example, vision, motor control, speech, and comprehension all require us to process high-dimensional inputs as they change over time. This is something that feed-forward nets are incredibly poor at modeling.

2.7.1 What is a Recurrent Neural Net?

One quite promising solution to tackling the problem of learning sequences of information is the recurrent neural network (RNN). RNNs are built on the same computational unit as the feed forward neural net, but differ in the architecture of how these neurons are connected to one another. Feed forward neural networks were organized in layers, where information flowed unidirectionally from input units to output units. There were no undirected cycles in the connectivity patterns. Although neurons in the brain do contain undirected cycles as well as connections within layers, we chose to impose these restrictions to simplify the training process at the expense of computational versatility. Thus, to create more powerful compuational systems, we allow RNNs to break these artificially imposed rules. RNNs do not have to be organized in layers and directed cycles are allowed. In fact, neurons are actually allowed to be connected to themselves.

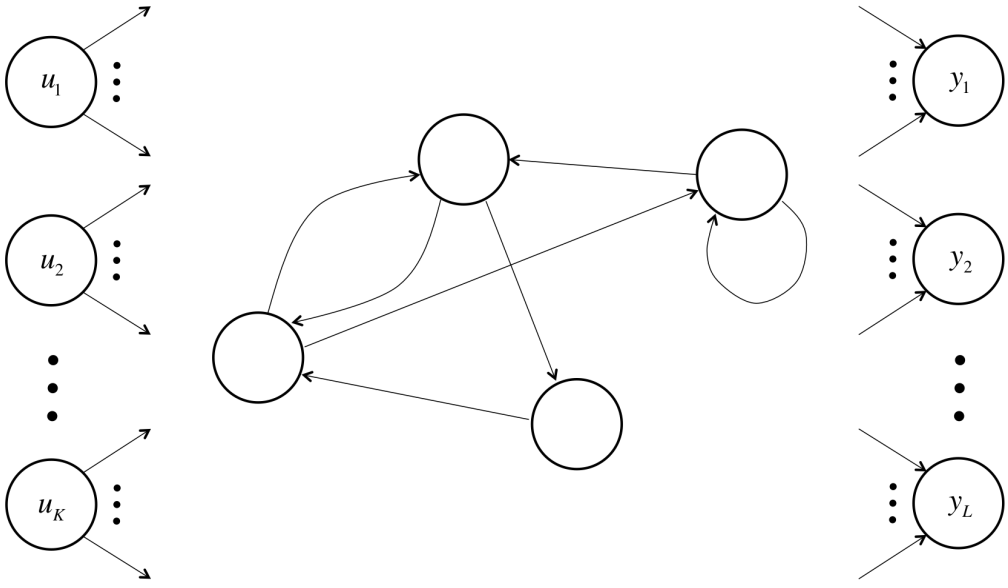


Fig. 2.23: An example schematic of a RNN with directed cycles and self connectivities.

The RNN consists of a bunch of input units, labeled $u_1, \cdots, u_K$, and output units, labeled $y_1, , y_L$. There are also the hidden units $x_1, \cdots, x_N$, which do most of the interesting work. You'll notice that the illustration shows a unidirectional flow of information from the input units to the hidden units as well as another unidirectional flow of information from the hidden units to the output units. In some cases, RNNs break the latter restriction with connections leading

from the output units back to the hidden units. These are called "backprojections," and don't make the analysis of RNNs too much more complicated. The same techniques we will discuss here will also apply to RNNs with backprojections.

There are a lot of pretty challenging technical difficulties that arise when training recurrent neural networks, and it's still a very active area of research. Hopefully by the end of this article, we'll have a solid understanding of how RNNs work and some of the results that have been achieved!

2.7.2 Simulating a Recurrent Neural Network

Now that we understand how a RNN is structured, we can discuss how it's able to simulate a sequence of events. Let's consider a neat toy example of a recurrent neural net acting like an timer module, a classic example designed by Herbert Jaeger (his original manuscript can be found in[1]).

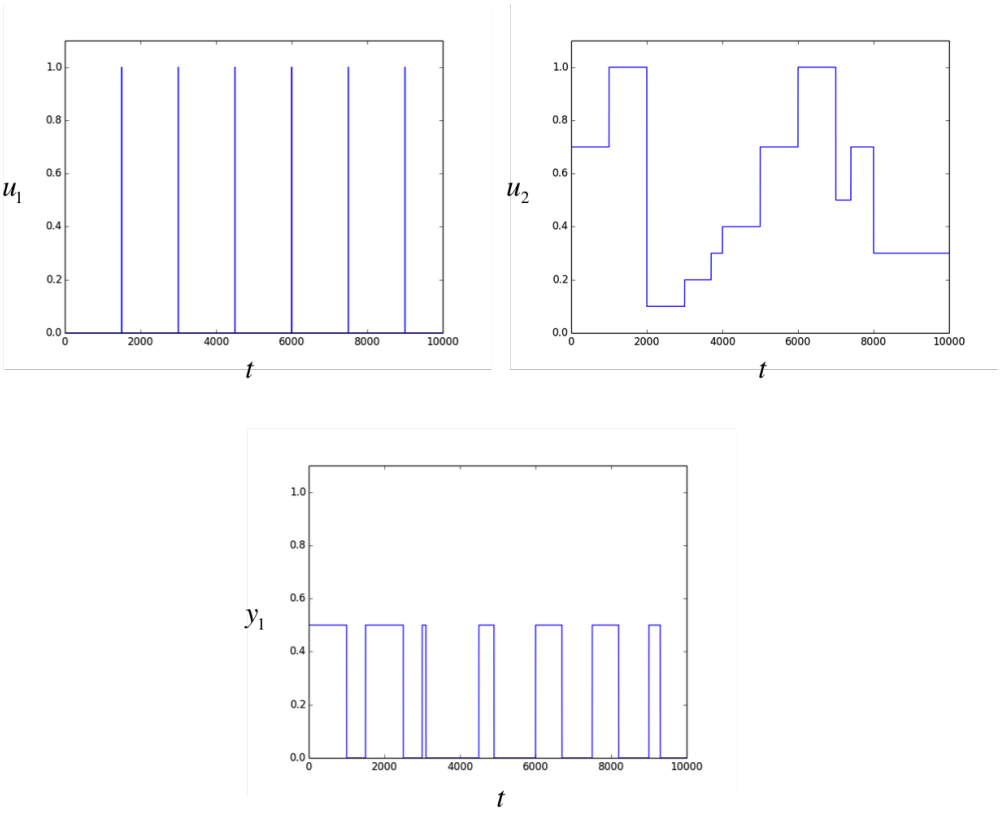


Fig. 2.24: A simple example of how a perfect RNN would simulate a timer.

In this case, we have two inputs. The input $u1$ corresponds to a binary switch which spikes to one when the RNN is supposed to start the timer. The input $u2$ is a discrete variable that varies between 0.1 and 1.0 inclusive which corresponds to how long the output should be turned on if the timer is started at that instant. The RNN's specification requires it to turn on the output for a duration of $1000u_2$. Finally, the outputs in the training examples toggle between 0 (off) and 0.5 (on).

But how exactly would a neural net achieve this calculation? First, the RNN has all of its hidden activities initialized to some pre-determined state. Then at each time step (time $t = 1, 2, \cdots$), every hidden unit sends its current activity through all its outgoing connections. It then recalculate its new activity by computing the weighted sum (logit) of

[1] Jaeger, Herbert. Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach. Vol. 5. GMD-Forschungszentrum Informationstechnik, 2002.

its inputs from other neurons (including itself if there is a self connection) and the current values of the inputs, and then feeding this value into a neuron-specific function (a straightforward copy operation, sigmoid transform, soft-max, etc.). Because the previous vector of activities is used to compute the vector of activies in each time step, RNNs are able to retain memory of previous events and utlize this memory in making decisions.

Clearly a neural net would be unlikely to perfectly perform according to specification, but you can imagine it outputting a result (orange) that looks pretty darn close to the ground truth (blue) after training the RNN with hundreds or thousands of examples. We'll talk more about approaches to training RNNs in the following sections.

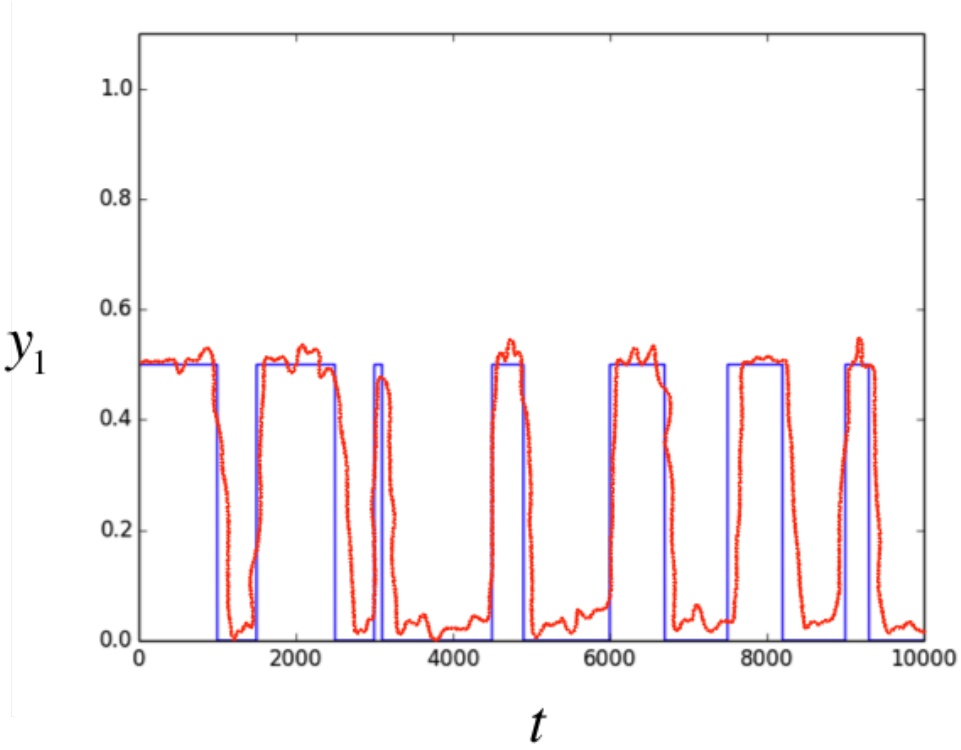


Fig. 2.25: An example fit for how a well-trained RNN might approximate the output of a test case.

At this point, you're probably thinking that this is pretty cool, but it's still a pretty contrived example. What's the strategy for using RNNs in practice? We examine real systems and their behaviors over time in response to stimuli. For example, you might teach a RNN to transcribe audio into text by building a dataset (in a sense, observing the response of the human auditory system in response to the inputs in the training set). You may also use a trained neural net to model a system's reactions under novel stimuli.

But if you're creative, you can use RNNs in ways that seem pretty spectacular. For example, a specialized kind of RNN, called a long short-term RNN or LSTM, has been used to achieve spectacular rates of data compression (although the current approach to RNN-based compression does take a significant amount time). For those itching to learn more, we'll talk about the LSTM architecture in a later section.

2.7.3 Training a RNN - Backpropagation Through Time

Great, now we understand what a RNN is and how it works, but how do we train a RNN in the first place to achieve all of these spectacular feats? Specifically, how do we determine the weights that are on each of the connections? And how do we choose the initial activities of all of the hidden units? Our first instinct might be to use backpropagation directly, after all it worked quite well when we used it on feed forward neural nets.

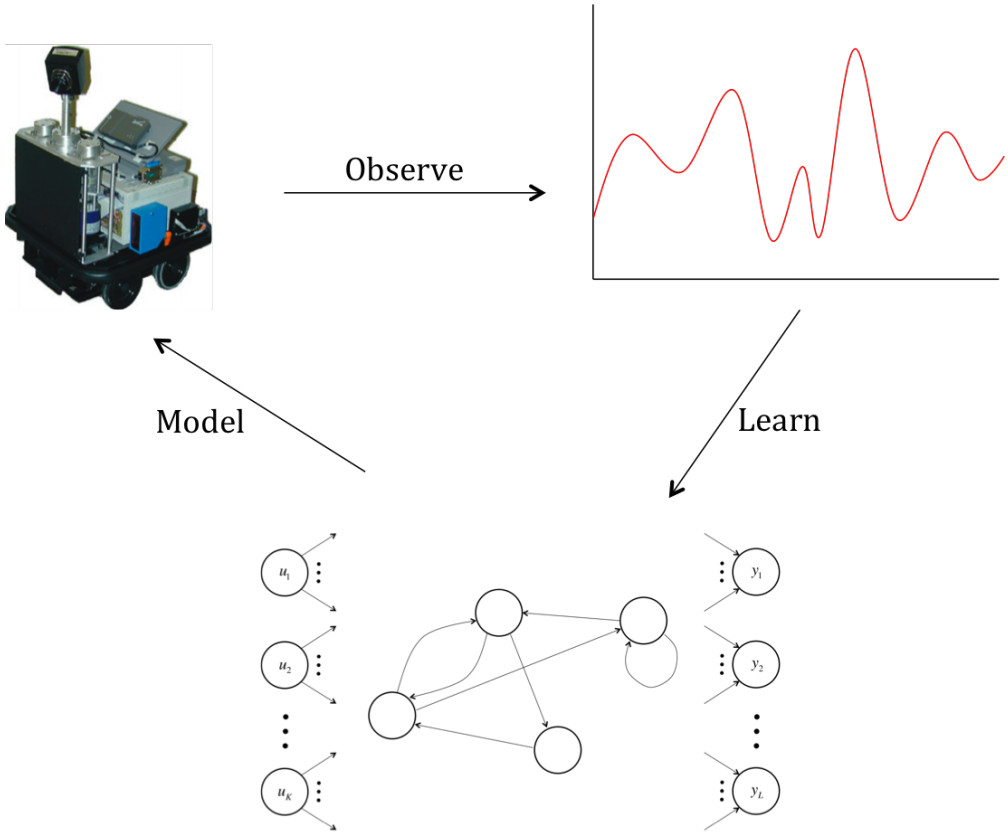


Fig. 2.26: How a RNN might be used in practice.

The problem with using backpropagation here is that we have cyclical dependencies. In feed forward nets, when we calculated the error derivatives with respect to the weights in one layer, we could express them completely in terms of the error derivatives from the layer above. In a recurrent neural network, we don't have this nice layering because the neurons do not form a directed acyclic graph. Trying to backpropagate through a RNN could force us to try to express an error derivative in terms of itself, which doesn't make for easy analysis.

So how can we use backpropagation for RNNs, if at all? The answer lies in employing a clever transformation, where we convert our RNN into a new structure that's essentially a feed-forward neural network! We call this strategy "unrolling" the RNN through time, and an example can be seen in the figure below (with only one input/output per time step to simplify the illustration):

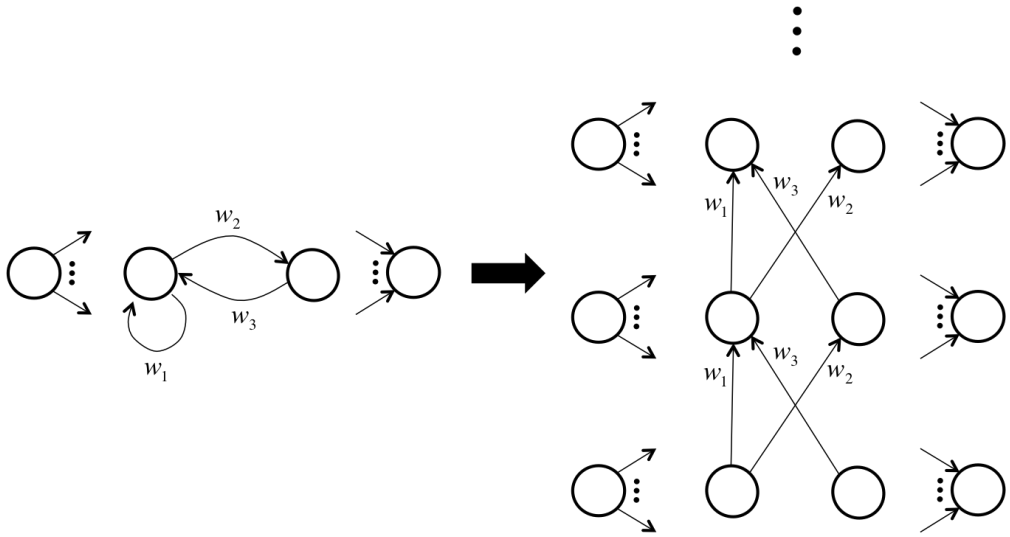


Fig. 2.27: An example of "unrolling" and RNN through time to use backpropagation.

The process is actually quite simple, but it has a profound impact on our ability to analyze the neural network. We take the RNN's inputs, outputs, and hidden units and replicate it for every time step. These replications correspond to layers in our new feed forward neural network. We then connect hidden units as follows. If the original RNN has a connection of weight w from neuron i to neuron j, in our feed forward neural net, we draw a connection of weight w from neuron i in every layer t_k to neuron j in every layer $t_k + 1$.

Thus, to train our RNN, we randomly initialize the weights, "unroll" it into a feed forward neural net, and backpropogate to determine the optimal weights! To determine the initializations for the hidden states at time 0, we can treat the initial activities as parameters fed into the feed forward network at the lowest layer and backpropagate to determine their optimal values as well!

We run into a problem however, which is that after every batch of training examples we use, we need to modify the weights based on the error derivatives we calculated. In our feed-forward net, we have sets of connections that all correspond to the same connection in the original RNN. The error derivatives calculated with respect to their weights, however, are not guaranteed to be equal, which means we might be modifying them by different amounts. We definitely don't want to be doing that!

We can get around this challenge, by averaging (or summing) the error derivatives over all the connections that belong to the same set. This means that after each batch, we modify corresponding connections by the same amount, so if they were initialized to the same value, they will end up at the same value. This solves our problem :)

2.7.4 The Problems with Deep Backpropagation

Unlike traditional feed forward nets, the feed forward nets generated by unrolling RNNs can be enormously deep. This gives rise to a serious practical issue: it can be obscenely difficult to train using the backpropagation through time approach. Let's take a step back and try to understand why.

Let's try to train a RNN to do a very primitive task. Let's give the RNN a single hidden unit with a bias term, and we'll connect it to itself and a singular output. We want this neural network to output a fixed target value after 50 steps, let's say 0.7. We'll use the squared error of the output on the 50th time step as our error function, which we can plot as a surface over the value of the weight and the bias:

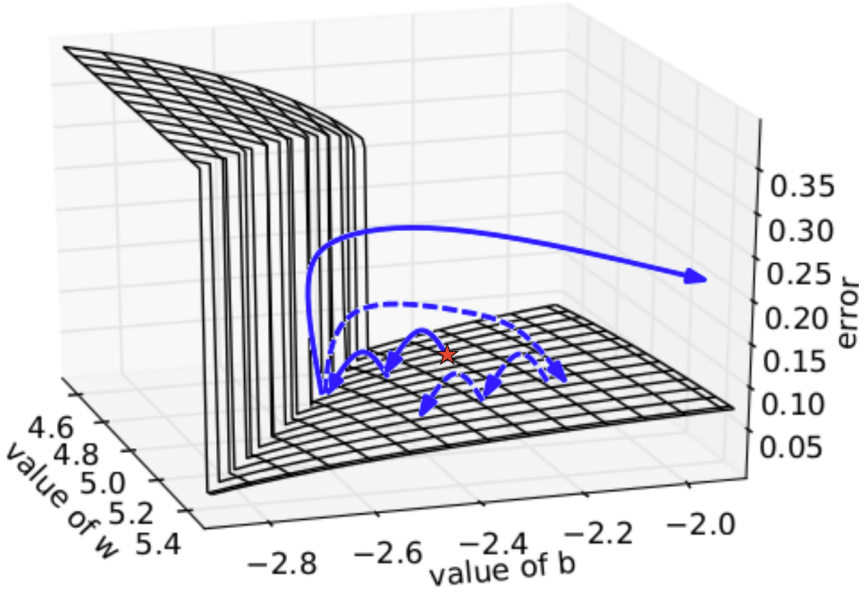


Fig. 2.28: The problematic error surface of a simple RNN.

Now, let's say we started at the red star (using a random initialization of weights). You'll notice that as we use gradient descent, we get closer and closer to the local minimum on the surface. But suddenly, when we slightly overreach the valley and hit the cliff, we are presented with a massive gradient in the opposite direction. This forces us to bounce extremely far away from the local minimum. And once we're in nowhere land, we quickly find that the gradients are so vanishingly small that coming close again will take a seemingly endless amount of time. This issue is called the problem of exploding and vanishing gradients. You can imagine perhaps controlling this issue by rescaling gradients to never exceed a maximal magnitude (see the dotted path after hitting the cliff), but this approach still doesn't perform spectacularly well, especially in more complex RNNs. For a more mathematical treatment of this issue, check out this paper.[2]

2.7.5 Long Short Term Memory

To address these problems, researchers proposed a modified architecture for recurrent neural networks to help bridge long time lags between forcing inputs and appropriate responses and protect against exploding gradients. The architecture forces constant error flow (thus, neither exploding nor vanishing) through the internal state of special memory units. This long short term memory (LSTM) architecture utlized units that were structured as follows:

The LSTM unit consists of a memory cell which attempts to store information for extended periods of time. Access to this memory cell is protected by specialized gate neurons - the keep, write, and read gates - which are all logistic units.

[2] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." ICML (3) 28 (2013): 1310-1318.

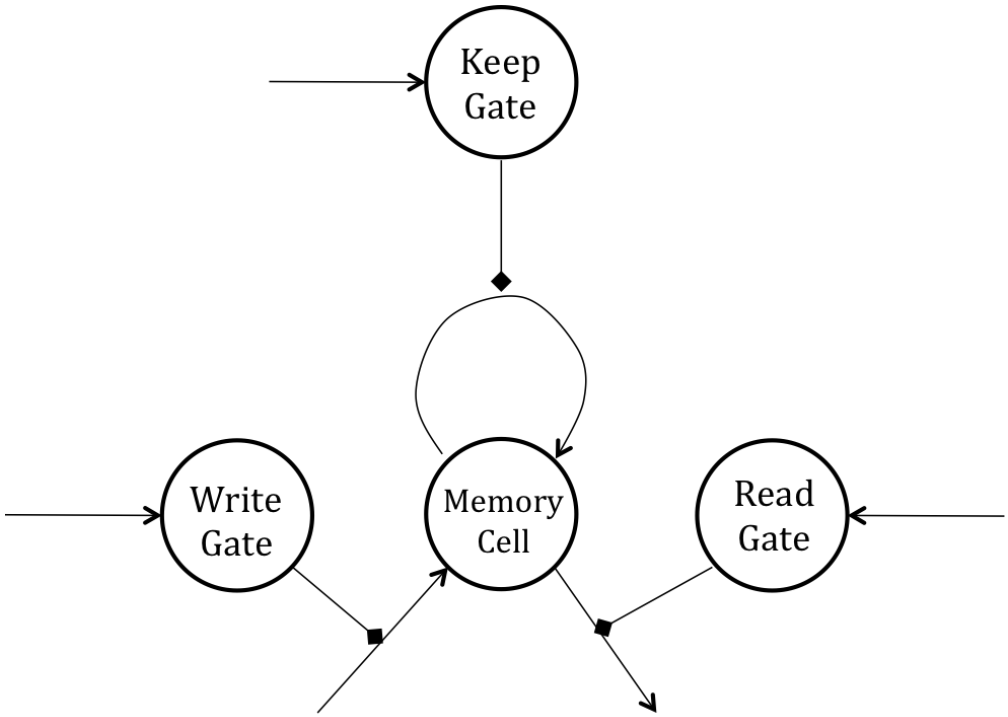


Fig. 2.29: Structure of the basic LSTM unit.

These gate cells, instead of sending their activities as inputs to other neurons, set the weights on edges connecting the rest of the neural net to the memory cell. The memory cell is a linear neuron that has a connection to itself. When the keep gate is turned on (with an activity of 1), the self connection has weight one and the memory cell writes its contents into itself. When the keep gate outputs a zero, the memory cell forgets its previous contents. The write gate allows the rest of the neural net to write into the memory cell when it outputs a 1 while the read gate allows the rest of the neural net to read from the memory cell when it outputs a 1.

So how exactly does this force a constant error flow through time to locally protect against exploding and vanishing gradients? To visualize this, let's unroll the LSTM unit through time:

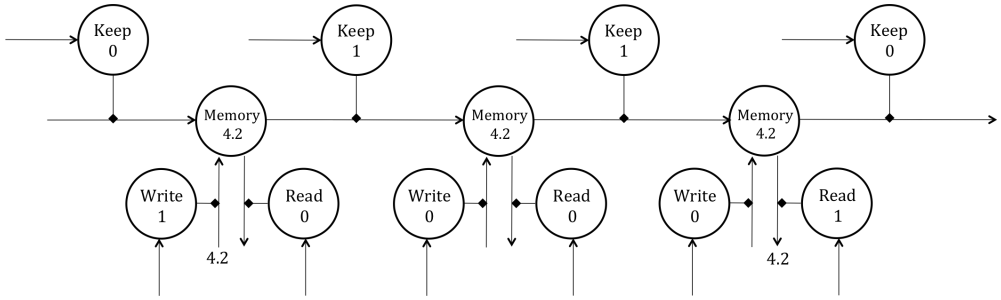


Fig. 2.30: Unrolling the LSTM unit through the time domain.

At first, the keep gate is set to 0 and the write gate is set to 1, which places 4.2 into the memory cell. This value is retained in the memory cell by a subsequent keep value of 1 and protected from read/write by values of 0. Finally, the cell is read and then cleared. Now we try to follow the backpropagation from the point of loading 4.2 into the memory cell to the point of reading 4.2 from the cell and its subsequent clearing. We realize that due to the linear nature of the memory neuron, the error derivative that we receive from the read point backpropagates with negligible change until the write point because the weights of the connections connecting the memory cell through all the time layers

have weights approximately equal to 1 (approximate because of the logistic output of the keep gate). As a result, we can locally preserve the error derivatives over hundreds of steps without having to worry about exploding or vanishing gradients. You can see the action of this method successfully reading cursive handwriting (Vedio).

The animation, borrowed from neural networks expert Alex Graves, requires a little bit of explanation:

1. Row 1: Shows when the letters are recognized
2. Row 2: Shows the states of some of the memory cells (Notice how they get reset when a character is recognized!)
3. Row 3: Shows the writing as it's being analyzed by the LSTM RNN
4. Row 4: This shows the gradient backpropagated to the inputs from the most active character of the upper softmax layer (This tells you which data points are providing the most influence on your current decision for the character)

The LSTM RNN does quite well, and it's been applied in lots of other places as well. As we discussed earlier, deep architectures for LSTM RNNs have been used to achieve pretty astonishing data compression rates. For those who are interested in learning more about this particular application for LSTM RNNs can check out this paper.[3]

2.7.6 Conclusions

How to effectively train neural nets remains an area of active research and has resulted in a number of alternative approaches, with no clear winner. The LSTM RNN architecture is one such approach to improving the training of RNNs. Another approach is to use a much better optimzer that can deal with exploding and vanishing gradients. Hessian-free optimization tries to detect directions with a small gradient, but even smaller curvature. This problem allows it to perform much better than naive gradient descent. A third approach involves a very careful initialization of the weights in hopes that it will allow us to avoid the problem of exploding and vanishing gradients in the first place (e.g. echo state networks, momentum based approaches).

[3] Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).

CHAPTER 3

API References

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

3.1 npdl.layers

3.1.1 Base Layers

class npdl.layers.**Layer**

The *Layer* class represents a single layer of a neural network. It should be subclassed when implementing new types of layers.

Because each layer can keep track of the layer(s) feeding into it, a network's output *Layer* instance can double as a handle to the full network.

backward(*pre_grad*, **args*, ***kwargs*)

calculate the input gradient

connect_to(*prev_layer*)

Propagates the given input through this layer (and only this layer).

Parameters **prev_layer** : previous layer

The previous layer to propagate through this layer.

forward(*input*, **args*, ***kwargs*)

Calculate layer output for given input (forward propagation).

classmethod from_json(*config*)

From configuration

grads

Get layer parameter gradients as calculated from backward().

param_grads

Layer parameters and corresponding gradients.

params
Layer parameters.

Returns a list of numpy.array variables or expressions that parameterize the layer.

Returns list of numpy.array variables or expressions

A list of variables that parameterize the layer

Notes

For layers without any parameters, this will return an empty list.

to_json()
To configuration

3.1.2 Core Layers

class npdl.layers.**Linear**(*n_out*, *n_in=None*, *init='glorot_uniform'*)
A fully connected layer implemented as the dot product of inputs and weights.

Parameters **n_out** : (int, tuple)

Desired size or shape of layer output

n_in : (int, tuple) or None

The layer input size feeding into this layer

init : (Initializer, optional)

Initializer object to use for initializing layer weights

backward(*pre_grad*, **args*, ***kwargs*)
Apply the backward pass transformation to the input data.

Parameters **pre_grad** : numpy.array

deltas back propagated from the adjacent higher layer

Returns numpy.array

deltas to propagate to the adjacent lower layer

forward(*input*, **args*, ***kwargs*)
Apply the forward pass transformation to the input data.

Parameters **input** : numpy.array

input data

Returns numpy.array

output data

class npdl.layers.**Dense**(*n_out*, *n_in=None*, *init='glorot_uniform'*, *activation='tanh'*)
A fully connected layer implemented as the dot product of inputs and weights. Generally used to implemenent nonlinearities for layer post activations.

Parameters **n_out** : int

Desired size or shape of layer output

n_in : int, or None

The layer input size feeding into this layer

activation : str, or npdl.activatns.Activation

Defaults to `Tanh`

init : str, or npdl.initializations.Initializer

Initializer object to use for initializing layer weights

backward(*pre_grad*, **args*, ***kwargs*)

Apply the backward pass transformation to the input data.

Parameters **pre_grad** : numpy.array

deltas back propagated from the adjacent higher layer

Returns numpy.array

deltas to propagate to the adjacent lower layer

forward(*input*, **args*, ***kwargs*)

Apply the forward pass transformation to the input data.

Parameters **input** : numpy.array

input data

Returns numpy.array

output data

class npdl.layers.**Softmax**(*n_out*, *n_in=None*, *init='glorot_uniform'*)

A fully connected layer implemented as the dot product of inputs and weights.

Parameters **n_out** : int

Desired size or shape of layer output

n_in : int, or None

The layer input size feeding into this layer

init : str, or npdl.initializations.Initializer

Initializer object to use for initializing layer weights

class npdl.layers.**Dropout**(*p=0.0*)

A dropout layer.

Applies an element-wise multiplication of inputs with a keep mask.

A keep mask is a tensor of ones and zeros of the same shape as the input.

Each *forward()* call generates an new keep mask stochastically where there distribution of ones in the mask is controlled by the keep param.

Parameters **p** : float

fraction of the inputs that should be stochastically kept.

forward(*input*, *train=True*, **args*, ***kwargs*)

Apply the forward pass transformation to the input data.

Parameters **input** : numpy.array

input data

train : bool

is inference only

Returns numpy.array

output data

3.1.3 Convolution Layers

class npdl.layers.**Convolution**(*nb_filter*, *filter_size*, *input_shape=None*, *stride=1*, *init='glorot_uniform'*, *activation='relu'*)

Convolution operator for filtering windows of two-dimensional inputs.

When using this layer as the first layer in a model, provide the keyword argument *input_shape* (tuple of integers, does not include the sample axis), e.g. *input_shape=(3, 128, 128)* for 128x128 RGB pictures.

3.1.4 Embedding Layer

class npdl.layers.**Embedding**(*embed_words=None*, *static=None*, *input_size=None*, *n_out=None*, *nb_batch=None*, *nb_seq=None*, *init='uniform'*)

3.1.5 Normalization Layer

class npdl.layers.**BatchNormal**(*epsilon=1e-06*, *momentum=0.9*, *axis=0*, *beta_init='zero'*, *gamma_init='one'*)

Batch normalization layer (Ioffe and Szegedy, 2014) *[R11]* .

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Parameters **epsilon** **small float > 0**

Fuzz parameter. npdl expects epsilon >= 1e-5.

axis : integer

axis along which to normalize in mode 0. For instance, if your input tensor has shape (samples, channels, rows, cols), set axis to 1 to normalize per feature map (channels axis).

momentum : float

momentum in the computation of the exponential average of the mean and standard deviation of the data, for feature-wise normalization.

beta_init : npdl.initializations.Initializer

name of initialization function for shift parameter, or alternatively, npdl function to use for weights initialization.

gamma_init : npdl.initializations.Initializer

name of initialization function for scale parameter, or alternatively, npdl function to use for weights initialization.

 # Input shape

Arbitrary. Use the keyword argument *input_shape* (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

 # Output shape

Same shape as input.

References

[R11]

3.1.6 Pooling Layers

class npdl.layers.**MeanPooling**(*pool_size*)
Average pooling operation for spatial data.

Parameters **pool_size** : tuple of 2 integers,

factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.

Returns 4D numpy.array

with shape *(nb_samples, channels, pooled_rows, pooled_cols)* if dim_ordering='th' or 4D tensor with shape: *(samples, pooled_rows, pooled_cols, channels)* if dim_ordering='tf'.

class npdl.layers.**MaxPooling**(*pool_size*)
Max pooling operation for spatial data.

Parameters **pool_size** : tuple of 2 integers,

factors by which to downscale (vertical, horizontal). (2, 2) will halve the image in each dimension.

Returns 4D numpy.array

with shape *(nb_samples, channels, pooled_rows, pooled_cols)* if dim_ordering='th' or 4D tensor with shape: *(samples, pooled_rows, pooled_cols, channels)* if dim_ordering='tf'.

3.1.7 Recurrent Layers

class npdl.layers.**Recurrent**(*n_out*, *n_in=None*, *nb_batch=None*, *nb_seq=None*, *init='glorot_uniform'*, *inner_init='orthogonal'*, *activation='tanh'*, *return_sequence=False*)
A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to tasks such as unsegmented connected handwriting recognition[R12]_ or speech recognition.[R13]_

Parameters **n_out** : int

hidden number

n_in : int or None

input dimension

nb_batch : int or None

batch size

nb_seq : int or None

sequent length

init : npdl.intializations.Initliazer

init function

inner_init : npdl.intializations.Initliazer

inner init function, between hidden to hidden

activation : npdl.activations.Activation

activation function

return_sequence : bool

return total sequence or not.

References

[R12], *[R13]*

class npdl.layers.**SimpleRNN**(***kwargs*)

Fully-connected RNN where the output is to be fed back to input.

$$o_t = tanh(U_t x_t + W_t o_{t-1} + b_t)$$

Parameters **output_dim: dimension of the internal projections and the final output.**

init: weight initialization function.

Can be the name of an existing function (str), or a npdl function.

inner_init: initialization function of the inner cells.

activation: activation function.

Can be the name of an existing function (str), or a npdl function.

return_sequence: if 'return_sequences', 3D 'numpy.array' with shape

(batch_size, timesteps, units) will be returned. Else, return 2D *numpy.array* with shape *(batch_size, units)*.

References

[R14]

class npdl.layers.**GRU**(*gate_activation='sigmoid'*, *need_grad=True*, ***kwargs*)

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014. Their performance on polyphonic music modeling and speech signal modeling was found to be similar to that of long short-term memory.[R15]_ They have fewer parameters than LSTM, as they lack an output gate.[R16]_

$$z_t = \sigma(U_z x_t + W_z h_{t-1} + b_z)$$

$$z_t = r_t = \sigma(U_r x_t + W_r h_{t-1} + b_r)$$

$$h_t = tanh(U_h x_t + W_h(s_{t-1} \odot r_t) + b_h)$$

$$s_t = (1 - z_t) \odot h_t + z_t \odot s_{t-1}$$

Parameters **gate_activation** : npdl.activations.Activation

Gate activation.

need_grad **bool**

If *True*, will calculate gradients.

References

[R15], *[R16]*

class npdl.layers.**LSTM**(*gate_activation='sigmoid'*, *need_grad=True*, *forget_bias_num=1*, ***kwargs*)

Bacth LSTM, support mask, but not support training.

Long short-term memory (LSTM) is a recurrent neural network (RNN) architecture (an artificial neural network) proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber *[R17]* and further improved in 2000 by Felix Gers et al.[R18]_ Like most RNNs, a LSTM network is universal in the sense that given enough network units it can compute anything a conventional computer can compute, provided it has the proper weight matrix, which may be viewed as its program.

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_f)$$

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_h)$$

$$g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot tanh(c_t)$$

Parameters **gate_activation** : npdl.activations.Activation

Gate activation.

need_grad **bool**

If *True*, will calculate gradients.

forget_bias_num : int

integer.

References

[R17], *[R18]*

class npdl.layers.**BatchLSTM**(*gate_activation='sigmoid'*, *need_grad=True*, *forget_bias_num=1*, ***kwargs*)

Batch LSTM, support training, but not support mask.

Parameters **gate_activation** : npdl.activations.Activation

Gate activation.

need_grad **bool**

If *True*, will calculate gradients.

forget_bias_num : int

integer.

References

[R19], *[R20]*

backward(*pre_grad*, *dcn=None*, *dhn=None*)
Backward propagation.

Parameters **pre_grad** : numpy.array

Gradients propagated to this layer.

dcn : numpy.array

Gradients of cell state at *n* time step.

dhn : numpy.array

Gradients of hidden state at *n* time step.

Returns numpy.array

The gradients propagated to previous layer.

connect_to(*prev_layer=None*)
Connection to the previous layer.

Parameters **prev_layer** : npdl.layers.Layer or None

Previous layer.

AllW : numpy.array

type	i	f	o	g
bias				
x2h				
h2h				

forward(*input*, *c0=None*, *h0=None*)
Forward propagation.

Parameters **input** : numpy.array

input should be of shape (nb_batch,nb_seq,n_in)

c0 : numpy.array or None

init cell state

h0 : numpy.array or None

init hidden state

Returns numpy.array

Forward results.

3.1.8 Shape Layers

class npdl.layers.**Flatten**(*outdim=2*)

Base Layers

Layer	The *Layer* class represents a single layer of a neural network.

Core Layers

Linear	A fully connected layer implemented as the dot product of inputs and weights.
Dense	A fully connected layer implemented as the dot product of inputs and weights.
Softmax	A fully connected layer implemented as the dot product of inputs and weights.
Dropout	A dropout layer.

Convolution Layers

Convolution	Convolution operator for filtering windows of two-dimensional inputs.

Embedding Layer

Embedding	

Normalization Layer

BatchNormal	Batch normalization layer (Ioffe and Szegedy, 2014) [R1] .

Pooling Layers

MeanPooling	Average pooling operation for spatial data.
MaxPooling	Max pooling operation for spatial data.

Recurrent Layers

Recurrent	A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle.
SimpleRNN	Fully-connected RNN where the output is to be fed back to input.
GRU	Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014.
LSTM	Bacth LSTM, support mask, but not support training.

Continued on next page

Table 3.7 – continued from previous page

BatchLSTM	Batch LSTM, support training, but not support mask.

Shape Layers

Flatten	

3.2 npdl.activations

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.

3.2.1 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.

3.2.2 Detailed Description

class npdl.activations.**Activation**

Base class for activations.

derivative(*input=None*)

Backward step.

Parameters **input** : numpy.array, optional.

If provide *input*, this function will not use *last_forward*.

forward(*input*)

Forward Step.

Parameters **input** : numpy.array

the input matrix.

class npdl.activations.**Sigmoid**

Sigmoid activation function.

derivative(*input=None*)

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, input=None):
        """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.
        """
        last_forward = self.forward(input) if input else self.last_forward
        return np.multiply(last_forward, 1 - last_forward)
```

class npdl.activations.**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(*input=None*)

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, input=None):
        """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.
        """
        last_forward = self.forward(input) if input else self.last_forward
        return 1 - np.power(last_forward, 2)
```

class npdl.activations.**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(*input=None*)

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, input=None):
        """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.
        """
        last_forward = input if input else self.last_forward
        res = np.zeros(last_forward.shape, dtype=get_dtype())
        res[last_forward > 0] = 1.
        return res
```

class npdl.activations.**Linear**

Linear activation function.

derivative(*input=None*)

Backward propagation. 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, input=None):
        """Backward propagation.
        The backward also return identity matrix.

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

class npdl.activations.**Softmax**

Softmax activation function.

derivative(*input=None*)

Backward propagation.

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, input=None):
        """Backward propagation.

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

class npdl.activations.**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(*input=None*)

Backward propagation.

Returns float32

The derivative of Elliot function.

forward(*input*)

Forward propagation.

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 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):
        """Forward propagation.

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

    def derivative(self, input=None):
        """Backward propagation.

        Returns
        -------
        float32
            The derivative of Elliot function.
        """
        last_forward = 1 + np.abs(input * self.steepness) if input else self.last_
↪forward
        return 0.5 * self.steepness / np.power(last_forward, 2)
```

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

Elliot symmetric sigmoid transfer function.

derivative(*input=None*)

Backward propagation.

Returns float32

The derivative of SymmetricElliot function.

forward(*input*)
Forward propagation.

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

    """

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

    def forward(self, input):
        """Forward propagation.

        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 = 1 + np.abs(input * self.steepness)
        return input * self.steepness / self.last_forward

    def derivative(self, input=None):
        """Backward propagation.

        Returns
        -------
        float32
            The derivative of SymmetricElliot function.
        """
        last_forward = 1 + np.abs(input * self.steepness) if input else self.last_
↪forward
        return self.steepness / np.power(last_forward, 2)
```

class npdl.activations.**SoftPlus**
Softplus activation function.

derivative(*input=None*)
Backward propagation.

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, input=None):
        """Backward propagation.

        Returns
        -------
        float32
            The derivative of Softplus function.
        """
        last_forward = np.exp(input) if input else self.last_forward
        return last_forward / (1 + last_forward)
```

class npdl.activations.**SoftSign**

SoftSign activation function.

derivative(*input=None*)

Backward propagation.

Returns float32

The derivative of SoftSign function.

forward(*input*)
Forward propagation.

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):
        """Forward propagation.

        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, input=None):
        """Backward propagation.

        Returns
        -------
        float32
            The derivative of SoftSign function.
        """
        last_forward = np.abs(input) + 1 if input else self.last_forward
        return 1. / np.power(last_forward, 2)
```

3.3 npdl.initializations

3.3.1 Initializers

Zero

Continued on next page

Table 3.10 – continued from previous page

`One`
`Uniform`
`Normal`
`Orthogonal`

3.3.2 Detailed Description

3.4 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.

3.4.1 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.activations.Softmax()))
>>> model.compile(loss=npdl.objectives.SCCE(), optimizer=npdl.optimizers.SGD(lr=0.
↪005))
```

3.4.2 Objectives

MeanSquaredError	Computes the element-wise squared difference between `targets` and `outputs`.
MSE	alias of *MeanSquaredError*
HellingerDistance	Computes the multi-class hinge loss between predictions and targets.
HeD	alias of *HellingerDistance*
BinaryCrossEntropy	Computes the binary cross-entropy between predictions and targets.
BCE	alias of *BinaryCrossEntropy*
SoftmaxCategoricalCrossEntropy	Computes the categorical cross-entropy between predictions and targets.
SCCE	alias of *SoftmaxCategoricalCrossEntropy*

3.4.3 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.

`backward`(*outputs*, *targets*)

Backward function.

Parameters **outputs, targets** : numpy.array

The arrays to compute the derivatives of them.

Returns numpy.array

An array of derivative.

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

class npdl.objectives.**MeanSquaredError**
Computes the element-wise squared difference between `targets` and `outputs`.

In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors or deviations—that is, the difference between the estimator and what is estimated. MSE is a risk function, corresponding to the expected value of the squared error loss or quadratic loss. The difference occurs because of randomness or because the estimator doesn't account for information that could produce a more accurate estimate. *[R24]*

The MSE is a measure of the quality of an estimator—it is always non-negative, and values closer to zero are better.

The MSE is the second moment (about the origin) of the error, and thus incorporates both the variance of the estimator and its bias. For an unbiased estimator, the MSE is the variance of the estimator. Like the variance, MSE has the same units of measurement as the square of the quantity being estimated. In an analogy to standard deviation, taking the square root of MSE yields the root-mean-square error or root-mean-square deviation (RMSE or RMSD), which has the same units as the quantity being estimated; for an unbiased estimator, the RMSE is the square root of the variance, known as the standard deviation.

Notes

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

References

[R24]

backward(*outputs*, *targets*)
MeanSquaredError backward propagation.

$$dE = p - t$$

Parameters **outputs, targets** : numpy.array

The arrays to compute the derivative between them.

Returns numpy.array

Derivative.

forward(*outputs*, *targets*)
MeanSquaredError forward propagation.

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

Parameters **outputs, targets** : numpy.array

The arrays to compute the squared difference between.

Returns numpy.array

An expression for the element-wise squared difference.

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

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

In probability and statistics, the Hellinger distance (closely related to, although different from, the Bhattacharyya distance) is used to quantify the similarity between two probability distributions. It is a type of f-divergence. The Hellinger distance is defined in terms of the Hellinger integral, which was introduced by Ernst Hellinger in 1909.[R25]_ *[R26]*

Notes

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

References

[R25], *[R26]*

backward(*outputs*, *targets*)
HellingerDistance forward propagation.

forward(*outputs*, *targets*)
HellingerDistance forward propagation.

Parameters **outputs** : numpy 2D array

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

targets : numpy 2D array

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!)

Returns numpy 1D array

An expression for the Hellinger Distance

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

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

Returns numpy array

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.

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

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.

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.

backward(*outputs*, *targets*)
SoftmaxCategoricalCrossEntropy backward propagation.

$$dE = p - t$$

Parameters **outputs** : numpy 2D array

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

targets : numpy 2D array

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

Returns numpy 1D array

forward(*outputs*, *targets*)
SoftmaxCategoricalCrossEntropy forward propagation.

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

Parameters **outputs** : numpy.array

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

targets : numpy.array

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

Returns numpy 1D array

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

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

3.5 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:

3.5.1 Examples

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

```
>>> import npdl
>>> from npdl.activations import ReLU
>>> from npdl.activations 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))
```

3.5.2 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

3.5.3 Detailed Description

class npdl.optimizers.**SGD**(**args*, ***kwargs*)
Stochastic Gradient Descent (SGD) updates

Generates update expressions of the form:

- `param := param - learning_rate * gradient`

class `npdl.optimizers.`**`Momentum`**(*momentum=0.9*, **args*, ***kwargs*)
Stochastic Gradient Descent (SGD) updates with momentum

Generates update expressions of the form:

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

Parameters **momentum** : float

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

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`**(*momentum=0.9*, **args*, ***kwargs*)
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 **momentum** : float

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

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`**(*epsilon=1e-06*, **args*, ***kwargs*)
Adagrad updates

Scale learning rates by dividing with the square root of accumulated squared gradients. See *[R33]* for further description.

Parameters **epsilon** : float

Small value added for numerical stability.

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 *[R34]*.

References

[R33], *[R34]*

class npdl.optimizers.**RMSprop**(*rho=0.9*, *epsilon=1e-06*, **args*, ***kwargs*)
RMSProp updates

Scale learning rates by dividing with the moving average of the root mean squared (RMS) gradients. See *[R35]* for further description.

Parameters **rho** : float

Gradient moving average decay factor.

epsilon : float

Small value added for numerical stability.

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

[R35]

class npdl.optimizers.**Adadelta**(*rho=0.9*, *epsilon=1e-06*, **args*, ***kwargs*)
Adadelta updates

Scale learning rates by the ratio of accumulated gradients to accumulated updates, see *[R36]* and notes for further description.

Parameters **rho** : float

Gradient moving average decay factor.

epsilon : float

Small value added for numerical stability.

decay : float

Decay parameter for the moving average.

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

[R36]

class npdl.optimizers.**Adam**(*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)
Adam updates

Adam updates implemented as in *[R37]*.

Parameters **beta1** : float

Exponential decay rate for the first moment estimates.

beta2 : float

Exponential decay rate for the second moment estimates.

epsilon : float

Constant for numerical stability.

Notes

The paper *[R37]* 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

[R37]

class npdl.optimizers.**Adamax**(*beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, **args*, ***kwargs*)
Adamax updates

Adamax updates implemented as in *[R38]*. This is a variant of of the Adam algorithm based on the infinity norm.

Parameters **beta1** : float

Exponential decay rate for the first moment estimates.

beta2 : float

Exponential decay rate for the second moment estimates.

epsilon : float

Constant for numerical stability.

References

[R38]

3.6 npdl.model

Linear stack of layers.

3.6.1 Detailed Description

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

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

3.7 npdl.utils

3.7.1 Data Utils

npdl.utils.**one_hot**(*labels*, *nb_classes=None*)
One-hot encoding is often used for indicating the state of a state machine. When using binary or Gray code, a decoder is needed to determine the state. A one-hot state machine, however, does not need a decoder as the state machine is in the nth state if and only if the nth bit is high.

A ring counter with 15 sequentially-ordered states is an example of a state machine. A one-hot implementation would have 15 flip flops chained in series with the Q output of each flip flop connected to the D input of the next and the D input of the first flip flop connected to the Q output of the 15th flip flop. The first flip flop in the chain represents the first state, the second represents the second state, and so on to the 15th flip flop which represents the last state. Upon reset of the state machine all of the flip flops are reset to 0 except the first in the chain which is set to 1. The next clock edge arriving at the flip flops advances the one hot bit to the second flip flop. The hot bit advances in this way until the 15th state, after which the state machine returns to the first state.

An address decoder converts from binary or gray code to one-hot representation. A priority encoder converts from one-hot representation to binary or gray code.

In natural language processing, a one-hot vector is a $1N$ matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word.

Parameters labels iterable

nb_classes : (iterable, optional)

Returns numpy.array

Returns a one-hot numpy array.

npdl.utils.**unhot**(*one_hot_labels*)
Get argmax indexes.

Parameters **one_hot_labels** : numpy.array

Returns numpy.array

Returns a unhot numpy array.

3.7.2 Random Utils

npdl.utils.**get_rng**()
Get the package-level random number generator.

Returns `numpy.random.RandomState` instance

The `numpy.random.RandomState` instance passed to the most recent call of *set_rng()*, or `numpy.random` if *set_rng()* has never been called.

npdl.utils.**set_rng**(*rng*)
Set the package-level random number generator.

Parameters **new_rng** : `numpy.random` or a `numpy.random.RandomState` instance

The random number generator to use.

npdl.utils.**set_seed**(*seed*)
Set numpy seed.

Parameters **seed** : int

npdl.utils.**get_dtype**()
Get data dtype `numpy.dtype`.

Returns str or numpy.dtype

npdl.utils.**set_dtype**(*dtype*)
Set numpy dtype.

Parameters **dtype** : str or numpy.dtype

Data Utils

one_hot	One-hot encoding is often used for indicating the state of a state machine.
unhot	Get argmax indexes.

Random Utils

get_rng	Get the package-level random number generator.
set_rng	Set the package-level random number generator.
set_seed	Set numpy seed.
get_dtype	Get data dtype `numpy.dtype`.

Continued on next page

Table 3.14 – continued from previous page

set_dtype	Set numpy dtype.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[R11] [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](https://arxiv.org/abs/1502.03167)

[R12] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber. A Novel Connectionist System for Improved Unconstrained Handwriting Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 31, no. 5, 2009.

[R13] H. Sak and A. W. Senior and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. Proc. Interspeech, pp338-342, Singapore, Sept. 2010

[R14] A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. http://arxiv.org/abs/1512.05287

[R15] Chung, Junyoung; Gulcehre, Caglar; Cho, KyungHyun; Bengio, Yoshua (2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". arXiv:1412.3555Freely accessible [cs.NE].

[R16] "Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano – WildML". Wildml.com. Retrieved May 18, 2016.

[R17] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". Neural Computation. 9 (8): 1735–1780. doi:10.1162/ne co.1997.9.8.1735. PMID 9377276.

[R18] Felix A. Gers; Jürgen Schmidhuber; Fred Cummins (2000). "Learning to Forget: Continual Prediction with LSTM". Neural Computation. 12 (10): 2451–2471. doi:10.1162/089976600300015015.

[R19] Sepp Hochreiter; Jürgen Schmidhuber (1997). "Long short-term memory". Neural Computation. 9 (8): 1735–1780. doi:10.1162/ne co.1997.9.8.1735. PMID 9377276.

[R20] Felix A. Gers; Jürgen Schmidhuber; Fred Cummins (2000). "Learning to Forget: Continual Prediction with LSTM". Neural Computation. 12 (10): 2451–2471. doi:10.1162/089976600300015015.

[R24] Lehmann, E. L.; Casella, George (1998). Theory of Point Estimation (2nd ed.). New York: Springer. ISBN 0-387-98502-6. MR 1639875.

[R25] Nikulin, M.S. (2001), "Hellinger distance", in Hazewinkel, Michiel, Encyclopedia of Mathematics, Springer, ISBN 978-1-55608-010-4

[R26] Jump up ^ Hellinger, Ernst (1909), "Neue Begründung der Theorie quadratischer Formen von unendlichvielen Veränderlichen", Journal für die reine und angewandte Mathematik (in German), 136: 210–271, doi:10.1515/crll.1909.136.210, JFM 40.0393.01

[R33] Duchi, J., Hazan, E., & Singer, Y. (2011): Adaptive subgradient methods for online learning and stochastic optimization. JMLR, 12:2121-2159.

[R34] Chris Dyer: Notes on AdaGrad. http://www.ark.cs.cmu.edu/cdyer/adagrad.pdf

[R35] 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)

[R36] Zeiler, M. D. (2012): ADADELTA: An Adaptive Learning Rate Method. arXiv Preprint arXiv:1212.5701.

[R37] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

[R38] Kingma, Diederik, and Jimmy Ba (2014): Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980.

Python Module Index

n

Index