10 Days Of Grad: Deep Learning From The First Principles (/neural-networks)

Day 1: Learning Neural Networks The Hard Way

Bogdan Penkovsky

Dec 11, 2018 · 12 min read · 10 Days Of Grad (http://penkovsky.com/categories/10-days-of-grad/)

Neural networks is a topic that recurrently appears throughout my life. Once, when I was a BSc student, I got obsessed with the idea to build an "intelligent" machine[1]. I spent a couple of sleepless nights thinking. I read a few essays shedding some light on this philosophical subject, among which the most prominent, perhaps, stand Marvin Minsky's writings[2]. As a result, I came across neural networks idea. It was 2010, and *deep learning* was not nearly as popular as it is now[3]. Moreover, no one made much effort linking to neural networks in calculus or linear algebra curricula. Even folks doing classical optimization and statistics sometimes seemed puzzled when hearing about neural nets. Some year later, I have implemented a simple neural net (https://bitbucket.org/masterdezign/labNN) with sigmoid activations as a part of a decision-making course. At the time I realized that existing state of our knowledge was still lacking to really build "thinking computers"[4].

It was 2012, the conference in Crimea, Ukraine where I attended a brilliant talk by Prof. Laurent Larger. He explained how to build a high-speed hardware for speech recognition using a laser. The talk really inspired me, and a year later I have started a PhD with an aim to develop *reservoir computing*, recurrent neural networks implemented directly in hardware. Finally, now I am using deep neural networks as a part of my job.

In this series of posts I will highlight some curious details of problem solving with neural networks. I am pretty much against duplicate efforts, therefore I would avoid repeating aspects described many times somewhere else. For those who are completely new to the subject, instead of introducing neural networks all over again I would refer to Chapter 1.2 of my PhD thesis (https://hal.archives-ouvertes.fr/tel-01591441/file/PhD_thesis-Penkovsky-arch.pdf). Here I will only summarize that neural networks are to some extent inspired by biological neurons. Similarly to its biological counterpart, artificial neuron receives many inputs, performs a nonlinear transformation, and produces an output. The equation below formalizes this kind of behavior:

$$y = f(w_1 x_1 + w_2 x_2 + \cdots + w_N x_N) = f(\sum_i w_i x_i), \quad (1)$$

where N is the number of inputs x_i, w_i are synaptic weights[5], and y is the result. Surprising as it may appear, in a modern neural network f can be practically any nonlinear function. This nonlinear function is often called an *activation function*. Congratulations, we have arrived at a neural network from 1950-s (https://en.wikipedia.org/wiki/Perceptron).

To continue reading this article, some mathematical background is useful, but not mandatory. Anyone can develop an intuition about neural networks!

A Word On Haskell

This series of posts practically illustrate all the concepts in Haskell programming language. To motivate you, here are some Q & A you always wanted to know.

Q:
(https://www.reddit.com/r/MachineLearning/comments/79l5y8/p_haskell_flexible_neural_networks_work_in/dp
Is there anything that makes Haskell particularly good for Neural Networks or are you simply doing it because you prefer to use Haskell?

A:
(https://www.reddit.com/r/MachineLearning/comments/79l5y8/p_haskell_flexible_neural_networks_work_in/dp
Neural networks are very "function objects". A network is just a big composition of functions. These kinds of things are very natural in a functional language.

Q:
(https://www.reddit.com/r/MachineLearning/comments/a57sqy/d_article_series_on_neural_networks/ebmlhrx)
As I am completely new to Haskell, would like to know what are the benefits of using Haskell vs python or other languages?

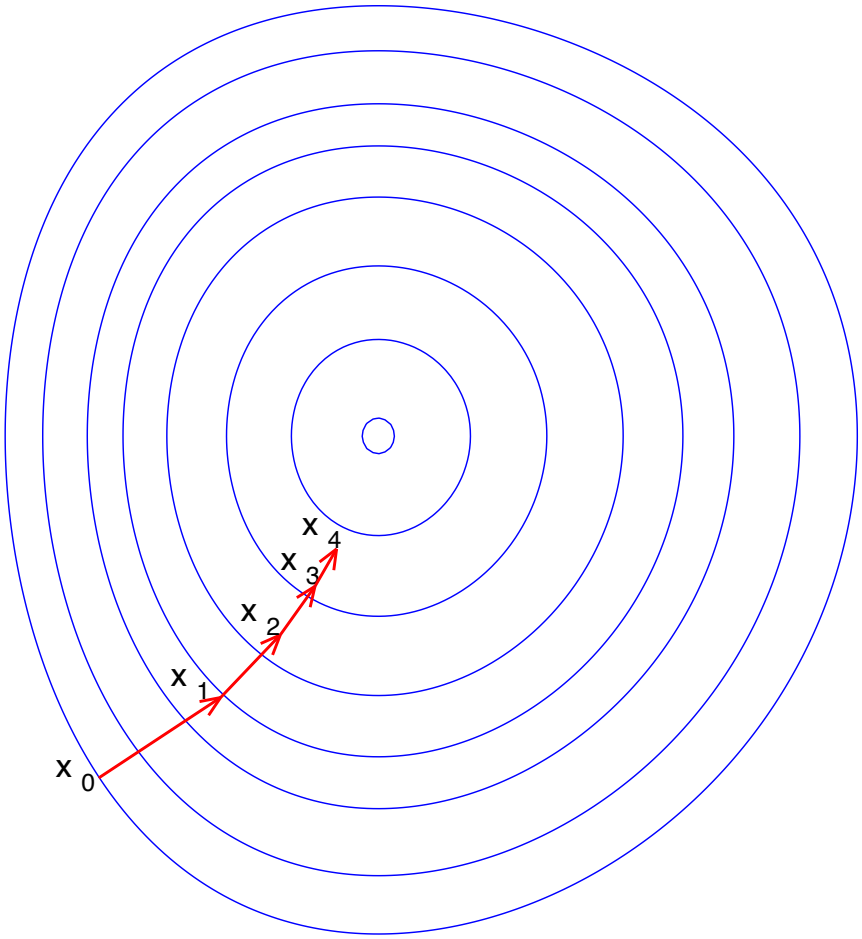
A: The benefits of using Haskell:

1. It is much easier to reason about what your program is doing.
2. Great when refactoring existing code base.
3. Haskell shapes your thinking towards problem solving in a pragmatic way.
4. Haskell programs are fast (https://stackoverflow.com/questions/35027952/why-is-haskell-ghc-so-darn-fast).

Q:
(https://www.reddit.com/r/MachineLearning/comments/a57sqy/d_article_series_on_neural_networks/ebkmzu9
I figured my lack of Haskell knowledge would make it hard to read, but your code examples still make sense to me.

A: Thank you. I find Haskell to be very intuitive when explaining neural nets.

Gradient Descent: A CS Freshman Year



Gradient Descent. Image credit (https://commons.wikimedia.org/wiki/File:Gradient_descent.svg)

The very basic idea behind neural networks training and deep learning is a local optimization method known as *gradient descent* (https://en.wikipedia.org/wiki/Gradient_descent). For those who have hard time remembering their freshman year, just watch an introductory video (https://www.youtube.com/watch?v=jc2IthslyzM) explaining the idea.

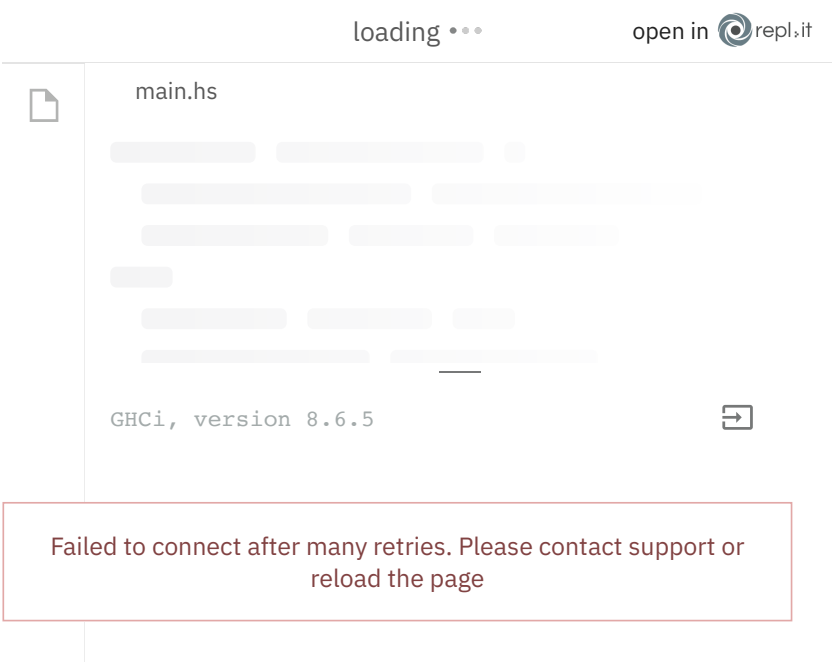
How does a concept as simple as gradient descent work for a neural network? Well, a neural network is only a function[6], mapping an input to some output. By comparing the neural network's output to some desired output, one can obtain another function known as an *error function*. This error function has a certain *error landscape*, like in the mountains. By using gradient descent, we modify our neural network in such a way that we are descending this landscape. Therefore, we aim at finding an error

minimum. The key concept of the optimization method is that the error gradients give us a direction in which to change the neural network. In a similar way one would be able to descend a hill covered with a thick fog. In both cases only a local gradient (slope) is available.

Gradient descent can be described by a formula:

$$x_{n+1} = x_n - \gamma \cdot \nabla F(x_n), \tag{2}$$

where constant γ is what is referred to in deep learning as the *learning rate*, i.e. the amount of learning per iteration n. In the simplest case, x is a scalar variable. The gradient descent method can be implemented in few lines of code. The Haskell code snippet (https://repl.it/@masterdezign/Grad) below is interactive. Press the green triangle button to Run the code.



That outputs the following sequence:
$0.0, 2.4, 2.88, 2.976, 2.9952, 2.99904, 2.999808, 2.9999616, \ldots$ Indeed, the value minimizing function $f(x)$ is 3, i.e. $\min f(x) = f(3) = (x-3)^2 = 0$. And the sequence is gradually converging towards that number. Let us look more carefully what does the code above do.

Lines 1-3: We define the gradient descent method, which iteratively applies the function `step` implementing equation (2). We provide the intermediate results taking the first `iterN` values.

Line 5: Suppose, we would like to optimize a function $f(x) = (x-3)^2$. Its gradient `gradF_test` is then $\nabla f(x) = 2 \cdot (x-3)$.

Lines 7-10: Finally, we run our gradient descent using learning rate $\gamma = 0.4$.

It is crucial to realize that the value of γ affects the convergence. When γ is too small, the algorithm will take many more iterations to converge, however, when γ is too large the algorithm will never converge. At the moment, I am not aware of a good way how to determine the best γ for a given problem. Therefore, often different γ values have to be tried. Feel free to modify the code above and see what comes out! For example, you can try out different values of `gamma`, such as `gamma = 0.01`, `gamma = 0.1`, `gamma = 1.1`. The method is generalizable to N dimensions. Essentially, we would replace the `gradF_test` function with the one operating on vectors rather than scalars.

Neural Network Ingredients For Classification

Iris plant. Image credit David Iliff (https://commons.wikimedia.org/wiki/File:Iris_germanica_(Purple_bearded_Iris),_Wakehurst_Place,_UK_-_Diliff.jpg) (CC-BY-SA 3.0) (https://creativecommons.org/licenses/by-sa/3.0/)

Now that we realize how the gradient descent works, we may want to train a moderately useful network. Let's say we want to perform *Iris* flower classification using four distinctive features[7]: sepal length, sepal width, petal length, petal width. There are three classes of flowers we want to be able to recognize: *Setosa*, *Versicolour*, and *Virginica*. Now, there is a problem: how do we encode those three classes so that our neural network can handle them?

Naive Solution And Why It Does Not Work

The most simple solution to indicate each species would be using natural numbers (https://en.wikipedia.org/wiki/Natural_number). For instance, *Iris Setosa* can be encoded as `1`, *Versicolour*, as `2`, and *Virginica*, as `3`. There is, however, a problem with this kind of encoding: we impose a *bias*. First, by encoding those classes as numbers, we impose a *linear order* over those three classes. It means, we start our count with *Setosa*, then, *Versicolour*, and then we arrive at *Virginica*. However, in reality it doesn't really matter if we end with *Virginica* or *Vernicolour*. Second, we also assume that the distance between *Virginica* and *Setosa* `3 - 1 = 2` is larger than between *Virginica* and *Versicolor* `3 - 2 = 1`, which is a priory wrong.

One-Hot Encoding

So which kind of encoding do we need? First, we want not to impose any restriction on ordering and second, we want the distances between classes to be equal. Therefore, we would prefer encoding each class to be orthogonal, i.e. independent from the other two. That becomes possible if we use vectors of three dimensions (as there are three classes). Therefore, now *Setosa* class is encoded as $[1, 0, 0]$, *Versicolour*, as $[0, 1, 0]$, and *Virginica* as $[0, 0, 1]$. The Euclidean distance (https://en.wikipedia.org/wiki/Euclidean_distance) between any pair of classes is equal to $\sqrt{2}$.
Update[8]: For example, the distance between *Setosa* and *Versicolour* is computed as

$$\sqrt{(1-0)^2 + (0-1)^2 + (0-0)^2} = \sqrt{2}.$$

Putting It All Together

Now that we are familiar with basic neural networks and gradient descent and also have some data to play with[7], let the fun begin!

First, we create a network of three neurons. To do that, we generalize formula (1):

$$y_i = f(\sum_k w_{ik} x_k), \tag{3}$$

where $(x_1, x_2, x_3, x_4) = \mathbf{x}$ is a 4D input vector, $w_{ik} \in \mathbf{W}, i = 1 \dots 3, k = 1 \dots 4$ is synaptic weights matrix, and result $(y_1, y_2, y_3) = \mathbf{y}$ is a 3D vector. Generally speaking, we perform a matrix-vector multiplication with a subsequent element-wise activation:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x}). \tag{4}$$

As a nonlinear activation function f we will use the *sigmoid* function[9] $\sigma(x) = [1 + e^{-x}]^{-1}$. We will exploit `hmatrix` (http://hackage.haskell.org/package/hmatrix) Haskell library for linear algebra operations such as matrix multiplication. With `hmatrix`, Equation (4) can be written as:

```
import Numeric.LinearAlgebra as LA

sigmoid = cmap f
  where
    f x = recip $ 1.0 + exp (-x)

forward x w =
  let h = x LA.<> w
      y = sigmoid h
  in [h, y]
```

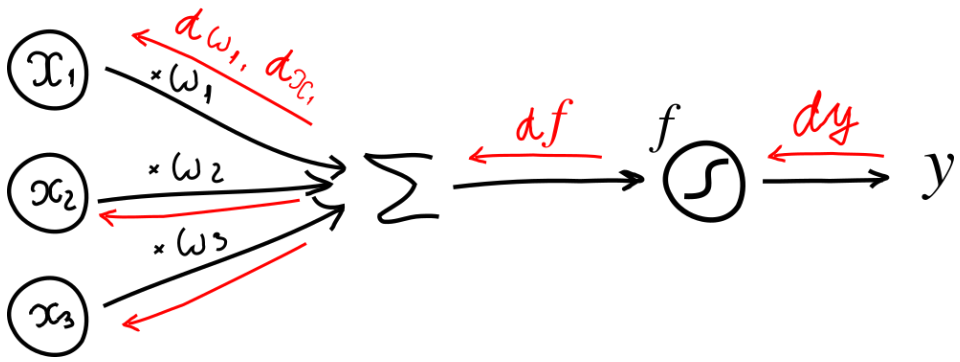
where `<>` denotes the matrix product function from `LA` module. Note that `x` can be a vector, but it can be also a dataset matrix. In the latter case, `forward` will transform our entire dataset. Notice that we provide not only the result of our computation `y`, but also an intermediate step `h` since it will be later reused for `w` gradient computation.

Each neuron y_i is supposed to fire when it 'thinks' that it has detected one of the three species. E.g. when we have an output [0.89, 0.1, 0.2], we would assume that the fist neuron is the most 'confident', i.e. we interpret the result as *Setosa*. In other words, this output is treated as similar to [1, 0, 0]. As you can see, the maximal element was set to one and others, to zero. This is a so-called 'winner takes all' rule.

Before training the neural network, we need some measure of error or *loss function* to minimize. For instance, we can use the Euclidean loss (http://arunmallya.github.io/writeups/nn/backprop.html#l2_loss_layer) $\text{loss} = \sum_i (\hat{y}_i - y_i)^2$ where $\hat{y}_i$ is a prediction and y_i is a real answer from our dataset:

```
loss y tgt =
  let diff = y - tgt
  in sumElements $ cmap (^2) diff
```

For the sake of gradient descend illustration, we will reuse the `descend` function defined above. Now, we have to specify the gradient function for our neural network equation (4). We use what is called a *backpropagation* or shortly *backprop* method, which is essentially a result (https://idontgetoutmuch.wordpress.com/2013/10/13/backpropogation-is-just-steepest-descent-with-automatic-differentiation-2/) of the chain rule (https://en.wikipedia.org/wiki/Chain_rule) and is illustrated for an individual neuron in the figure below[10].



Backpropation for a single neuron.
First, in the forward pass, initial output y is calculated. Then, this output is compared to some desired output and the error gradient dy is passed back. Afterwards, the activation function gradient df is obtained using dy. That ultimately leads to the remaining gradients $dw_1, dx_1, dw_2, dx_2, \ldots$.

Now we can calculate the weights gradient `dW` using the *backprop* method from above:

```
grad (x, y) w = dW
  where
    [h, y_pred] = forward x w
    dE = loss' y_pred y
    dY = sigmoid' h dE
    dW = linear' x dY
```

Here `linear'`, `sigmoid'`, `loss'` are gradients of linear operation (multiplication), sigmoid activation $\sigma(x)$, and the loss function. Note that by operating on matrices rather than scalar values we calculate the gradients vector `dW` denoting every synaptic weight gradient dw_i. Below are those "*vectorized* (https://en.wikipedia.org/wiki/Array_programming)" functions definitions in Haskell using `hmatrix` library[11]:

```
linear' x dy = cmap (/ m) (tr' x LA.<> dy)
  where
    m = fromIntegral $ rows x

sigmoid' x dY = dY * y * (ones - y)
  where
    y = sigmoid x
    ones = (rows y) >< (cols y) $ repeat 1.0

loss' y tgt =
  let diff = y - tgt
  in cmap (* 2) diff
```

To test our network, we download the dataset here (https://www.kaggle.com/masterdezign/iris-with-onehotencoded-targets) (there are two files: x.dat and y.dat) and the code here (https://gist.github.com/masterdezign/34ab610715df7dbf504cbe7cacdba68e). As instructed in the comments, we run our program:

```
$ stack --resolver lts-10.6 --install-ghc runghc --package hmatrix-0.18.2.0 Iris.hs
Initial loss 169.33744797846379
Loss after training 61.41242708538934
Some predictions by an untrained network:
(5><3)
[ 8.797633210095851e-2, 0.15127581829026382, 0.9482351750129188
,  0.11279346747947296,  0.1733431584272155, 0.9502442520696124
,  0.10592462402394615, 0.17057190568339017, 0.9367875655363787
,  0.10167941966201806, 0.20651101803783944, 0.9300343579182122
, 8.328154248684484e-2, 0.15568011758813116,  0.940816298954776 ]
Some predictions by a trained network:
(5><3)
[ 0.6989749292681016, 0.14916793398555747,  0.1442697900857393
,  0.678406436711954,  0.1691062984304366,  0.2052955124240905
, 0.6842327447503195, 0.16782087736820395, 0.16721778476233148
, 0.6262988163006756, 0.19656943129188192, 0.17521133197774072
, 0.6905553549763312, 0.15299944611286123, 0.12910826989854146 ]
Targets
(5><3)
[ 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0 ]
```

That's all for today. Please feel free to play with the code (https://gist.github.com/masterdezign/34ab610715df7dbf504cbe7cacdba68e). **Hint**: you may have noticed that `grad` calls `sigmoid` twice on the same data: once in `forward` and once in `sigmoid'`. Try optimizing the code to avoid this redundancy.

As soon as you understand the basics of neural networks, make sure you continue to Day 2 (/neural-networks/day2/). In the next post (/neural-networks/day2/) you will learn how to make your neural network operational. First of all, we will highlight the importance of multilayer structure. We will also show that nonlinear activations are crucial. Finally, we will improve neural network training and discuss weights initialization.

1. There even exists a term (https://en.wikipedia.org/wiki/Artificial_general_intelligence) describing exactly what I dreamed to achieve. ^
2. For instance, check Why people think computers can't (https://web.media.mit.edu/~minsky/papers/ComputersCantThink.txt). ^
3. With Google Trends in hand, we can witness the raise of global deep learning (https://trends.google.com/trends/explore?date=all&q=deep%20learning) interest. ^
4. Despite people have fantasized about it long before Alan Turing. ^
5. Essentially, a synaptic weight w_i determines the strength of connection to the i-th input. ^
6. With recurrent neural networks it is not true. Those have an internal state, making such networks equivalent to computer programs, i.e. potentially more complex than maps. ^
7. Here we refer to the classical Iris dataset (https://archive.ics.uci.edu/ml/datasets/iris). If you prefer another light dataset please let me know. ^
8. Kudos to Peter Harpending who spotted a typo in Euclidean distance. ^
9. In this example nonlinear activation is not essential. However, as we will see in future posts, in multilayer neural networks nonlinear activations are strongly required. ^
10. We look closer at the backpropagation mechanics here (/neural-networks/day3/). ^
11. And here (http://arunmallya.github.io/writeups/nn/backprop.html) are their mathematical derivations. ^

Deep Learning (http://penkovsky.com/tags/deep-learning/) Haskell (http://penkovsky.com/tags/haskell/)

Next: Day 2: What Do Hidden Layers Do? (http://penkovsky.com/neural-networks/day2/)

Related

- Towards Binarized Neural Networks Hardware (/talk/icee2018/)
- Delay Differential Equations (/project/dde/)
- HMEP (/project/hmep/)