[Jonathan Hui blog](https://jhui.github.io/)

[About](https://jhui.github.io/about/)

“Deep learning without going down the rabbit holes.”

March 18, 2017

**This is a work in progress…**

What is deep learning (DL)?

**Deep learning is about building a function estimator.** Historically, people explain deep learning (DL) using the neural network. Here, deep learning gains insight. Nevertheless, deep learning has outgrown this explanation. Once you realize building a deep learning network is about building a function estimator, you will unveil its real potential in Artificial Intelligence (AI).

Let’s build a new android named Pieter. Our first task is to teach Pieter to recognize objects visually. Can the human visual system be replaced by a big function estimator? Can we pass the pixel values to a function and classify it as a school bus, an airplane or a truck?

Indeed, in our later example of visual recognition, we will build a system very similar to the following:

For every node, we compute:

*z* *j* =∑ *i* *W* *ij* *x* *i* +*b* *i*  zj=∑iWijxi+bi

*f*(*z* *j* )=11+*e* −*z* *j*    f(zj)=11+e−zj

which *x* *i*  xi is the input pixel values or the inputs to a node.

*x* *i*  xi is called the **feature** in deep learning. **A deep network extracts features in the training data to make predictions.** If one of the nodes detects the amount of yellow color, it definitely helps us to differentiate a school bus from a shuttle bus.

Deep learning has many scary looking equations. We will walk through examples to show how it works. Most of them are pretty simple.

For example, with weight W (0.3, 0.2, 0.4, 0.3), bias b (-0.8) and a gray scale image with just 4 pixels (0.1, 0.3, 0.2, 0.1), the output of the first node circled in red above will be:

*z* *j* =0.3∗0.1+0.2∗0.3+0.4∗0.2+0.3∗0.1−0.8=−0.6 zj=0.3∗0.1+0.2∗0.3+0.4∗0.2+0.3∗0.1−0.8=−0.6

*f*(*z*)=11+*e* −(−0.6)  =0.3543 f(z)=11+e−(−0.6)=0.3543

Each node has its own weight (W) and bias (b). From the left most layer, we compute the output of each node and feed it to the next layer. Eventually, the right most layer is the probability for each object classification (a school bus 0.88, an airplane 0.08 or a truck 0.04). In this exercise, we supply all the weight and bias values to our android Pieter. But as the term “deep learning” implies, Pieter will learn those parameters by himself by the end of this tutorial. We still miss a few pieces for the puzzle, but the network diagram and the equations shown above lay down the foundation of a deep learning network. In fact, this simple design can recognize the zip code written on a envelope with very high accuracy.

XOR

For the skeptics, we will build an exclusive “or” (a xor b) using a simple network like:

For each node, we apply the same equations mentioned previously:

*z* *j* =∑ *i* *W* *i* ∗*x* *i* +*b* *i*  zj=∑iWi∗xi+bi

*h* *j* =*σ*(*z*)=11+*e* −*z* *j*    hj=σ(z)=11+e−zj

The following code implementation is self-explanatory. This demonstrates exactly how we do the weight multiplication and apply the sigmoid function. In this program, we use Numpy, a package for scientific computing with Python. It provides the many needed mathematical operations and array manipulations.

We provide coding to help the audience verify their understanding. Nevertheless, a full understanding of this code is not needed or suggested.

import numpy as np

**def** **sigmoid**(x):

**return** 1 **/** (1 **+** np**.**exp(**-**x))

**def** **layer1**(x1, x2):

w11, w21, b1 **=** 20, 20, **-**10

w12, w22, b2 **=** **-**20, **-**20, 30

v **=** w11**\***x1 **+** w21**\***x2 **+** b1

h11 **=** sigmoid(v)

v **=** w12**\***x1 **+** w22**\***x2 **+** b2

h12 **=** sigmoid(v)

**return** h11, h12

**def** **layer2**(x1, x2):

w1, w2, bias **=** 20, 20, **-**30

v **=** w1**\***x1 **+** w2**\***x2 **+** bias

**return** sigmoid(v)

**def** **xor**(a, b):

h11, h12 **=** layer1(a, b)

**return** layer2(h11, h12) *# Feed the output of last layer to the next layer*

**print**(" 0 ^ 0 = %.2f" **%** xor(0, 0)) *# 0.00*

**print**(" 0 ^ 1 = %.2f" **%** xor(0, 1)) *# 1.00*

**print**(" 1 ^ 0 = %.2f" **%** xor(1, 0)) *# 1.00*

**print**(" 1 ^ 1 = %.2f" **%** xor(1, 1)) *# 0.00*

The XOR output matches its expected logical values:

0 ^ 0 = 0.00

0 ^ 1 = 1.00

1 ^ 0 = 1.00

1 ^ 1 = 0.00

Delta function

Back to basic calculus, a function can be constructed with infinite narrow rectangles (a.k.a. delta function). If we construct such rectangles with a network, we can build on top of it to construct any functions.

Here is the code using the same set of equations and network layout before:

import numpy as np

import matplotlib.pyplot as plt

**def** **sigmoid**(x):

**return** 1 **/** (1 **+** np**.**exp(**-**x))

**def** **layer1**(x):

h11 **=** sigmoid(1000 **\*** x **-** 400)

h12 **=** sigmoid(1000 **\*** x **-** 500)

**return** h11, h12

**def** **layer2**(v1, v2):

**return** sigmoid(0.8 **\*** v1 **-** 0.8 **\*** v2)

**def** **func\_estimator**(x):

h11, h12 **=** layer1(x)

**return** layer2(h11, h12)

x **=** np**.**arange(0, 3, 0.001)

y **=** func\_estimator(x)

plt**.**plot(x, y)

plt**.**show()

This code produces an output shaped like a delta function.

Implementing an XOR or a delta function is not important for deep learning (DL). Nevertheless, we demonstrate the possibilities of building a complex function estimator through a network of simple computation nodes. A three-layer network can implement a hand written recognition system for numbers with an accuracy of 95+%. **The deeper a network; the more complex it is.** For example, Microsoft ResNet (2015) for visual recognition contains 151 layers. Many modern models include 10 million tunable parameters. For many AI problems, the model needed to solve the problem is very complex. In autonomous driving, we can model a policy (turn, accelerate, or brake) to approximate what a human will do based on what they see in front of them. This policy is too difficult to model analytically. Alternatively, a deep learning network can be trained well enough to approximate the accuracy of a regular driver.

If we cannot solve a problem analytically, then train a model empirically.

Autonomous driving involves many aspects of AI. DL provides a model estimator that cannot be created analytically.

Build a Linear Regression Model

**Deep learning acquires its knowledge from training data.** We will demonstrate how Pieter learns the model parameters *W* W by processing training data. For example, Pieter wants to expand his horizon and start online dating. He wants to determine the relationship between the number of online dates and the years of education and the monthly income. Pieter starts with a simple linear model as follows:

number of dates=*W* 1 ∗years in school+*W* 2 ∗monthly income+*bias* number of dates=W1∗years in school+W2∗monthly income+bias

He surveys 1000 people in different communities and collects the information on their income, education and their number of online dates. Pieter wants to know how each community values intellectual versus his humble post-doc salary. So, even though this model looks overwhelmingly simple, it serves its purpose. So, the task for Pieter is to find the parameter values W and b in this model using the training data collected by him.

The steps include:

1. Take the first guess on W and b.
2. Use the model to predict the number of dates for each sample in the training dataset.
3. Compute the mean square error between the computed value and the true value in the dataset.
4. Compute how much the error will change when we change W and b.
5. Re-adjust W & b according to this error rate change. (**Gradient descent**)
6. Back to step 2 for N iterations.
7. Use the last value of W & b for our model.

We build a model for each community and use these models to predict how well Pieter may do in each community. In our model, we predict the number of dates for people with certain income and years of education. The corresponding values (the number of dates) in the training dataset are called the **true values or true labels**.

Gradient descent

**Deep learning is about learning how much it costs.** Step 2-5 is called the gradient descent in DL. We define a function to measure errors between our model and the true values. In DL, this error function is called **cost function** or **loss function**. Mean square error (MSE) is one obvious candidate.

mean square error=*J*(*h*,*y*,*W*,*b*)=1*N* ∑ *i* (*h* *i* −*y* *i* ) 2  mean square error=J(h,y,W,b)=1N∑i(hi−yi)2

where *h* *i*  hi is the model prediction and *y* *i*  yi is the true value for sample *i* i . We add all the samples and take the average. We can visualize the cost below with x-axis being *W* 1  W1 and y-axis being *W* 2  W2 and z-axis being the cost J. The solution of our model is to find *W* 1  W1 and *W* 2  W2 where the cost is lowest. We can visualize this as dropping a marble at a random point (*W* 1 ,*W* 2 ) (W1,W2) and then let gravity do its work.

Optimizing a deep network means finding all the *W* W , *b* b and other tunable parameters to minimize cost.

Learning rate

Thinking in 3D or high dimensions is difficult, in not impossible. Try to think of DL problems in 2D first. Consider a point at (L1, L2) where we cut through the diagram along the blue and orange line then plot those curves in a 2D diagram.

The x-axis is *W* W and the y-axis is the cost. Since we are holding *W* 2  W2 as a constant, we can ignore it in the equation below to simplify the discussion.

*J*(*W*,*b*,*h*,*y*)=1*N* ∑ *i* (*W* 1 ∗*x* *i* −*y* *i* ) 2  J(W,b,h,y)=1N∑i(W1∗xi−yi)2

**Training a model with gradients**. Since the gradient at L1 is negative, we move *W* 1  W1 to the right to find the lowest point. But by how much? L2 has a smaller gradient than L1. So, changing *W*2 W2 has a smaller impact on cost in comparison to L1. Obviously, we should update a parameter proportional to its impact. Therefore, adjustment for (*W* 1 ,*W* 2 ) (W1,W2) is proportional to its partial gradient at that point. i.e.

Δ*W* *i* ∝∂*J*∂*W* *i*   ΔWi∝∂J∂Wi

 i.e. Δ*W* 1 ∝∂*J*∂*W* 1   and Δ*W* 2 ∝∂*J*∂*W* 2    i.e. ΔW1∝∂J∂W1 and ΔW2∝∂J∂W2

Add a ratio value, the adjustments to *W* W are:

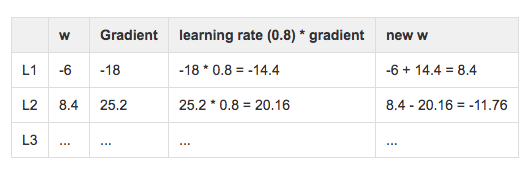
Δ*W* *i* =*α*∂*J*∂*W* *i*   ΔWi=α∂J∂Wi

*W* *i* =*W* *i* −Δ*W* *i*  Wi=Wi−ΔWi

The variable *α* α is called the **learning rate**. A **small learning rate learns slowly.** A small learning rate takes a longer time or more reptitions to locate the minimum. However, as we learn in calculus, a larger step results in a larger error in the calculation. In DL, finding the right value for the learning rate is a trial and error exercise. We usually try values ranging from 1e-7 to 1 in logarithmic scale (1e-7, 5e-7, 1e-6, 5e-6, 1e-5 …), but this depends on the problem you are solving. Other parameters such as learning rate need to be tuned. We call all these parameters **“hyperparameters”**.

A large learning step may have other serious problems. It costs *w* w to oscillate with increasing cost:

We start with w = -6 (x-axis) at L1. If the gradient is huge, the learning rate is larger than a certain value and will swing *w* w too far to the other side (say L2) creating an even larger gradient. Eventually, rather than dropping down slowly to a minimum, *w* w oscillates and the cost increases. When loss keeps going upward, we need to reduce the learning rate. The following demonstrates how a learning rate of 0.8 with a steep gradient swings the cost upward instead of downward. The table traces how the oscillation of W causes the cost to go upwards from L1 to L2 and then L3.



We need to be careful about the scale used for the x-axis and y-axis. In the diagram above, the gradient does not look steepbecause we have a much smaller scale fory-axis is 0 to 150 while the x-axis is -10 to 10.

**A large learning rate overshoots your target.** Here is another illustration of some real problems. When we gradually descend, we may land in an area with a steep gradient in which the *W* W bounces back. With this shape it is very difficult to find the minimum with a constant learning rate. Advanced methods to address this problem will be discussed later.

This example is dramatic, but real. But in a lesser extent, instead of settling down at the bottom, *W* W oscillates around the minimum slightly. If we drop a ball in the Grand Canyon, we expect it to land in the bottom. In DL, this is more difficult.

Adjusting learning rate with better optimization techniques is heavily studied and is still in active research for very complex problems.

Naive gradient checking

There are many ways to compute a partial derivative. One naive but important method is using the simple partial derivative definition.

∂*f*∂*x* =*f*(*x*+Δ*x* *i* )−*f*(*x*−Δ*x* *i* )2Δ*x* *i*   ∂f∂x=f(x+Δxi)−f(x−Δxi)2Δxi

Here is a simple code demonstrating the derivative of *x* 2  at *x*=4 x2 at x=4

**def** **gradient\_check**(f, x, h**=**0.00001):

grad **=** (f(x**+**h) **-** f(x**-**h)) **/** (2**\***h)

**return** grad

f **=** **lambda** x: x**\*\***2

**print**(gradient\_check(f, 4))

We never use this method in production; however, computing partial derivative is tedious and error prone. We use the naive method to verify a partial derivative implementation during development.

Mini-batch gradient descent

When computing the cost function, we can add all the errors for the entire training dataset. This takes too much time for just one update in one repetition. On the contrary, we can perform stochastic gradient descent which makes one *W* W update per training sample. Nevertheless, the gradient descent will follow a zip zap pattern rather than follow the curve of the cost function. This can be a problem if you land in a steep gradient area where the parameters bounce to an area with a high cost. Stochastic gradient descent takes longer and may zip zag around the minimum rather than converge to it.

A good compromise is to process a batch of N samples at a time. N is a tunable hyperparameter, but usually is not very critical. We can start with 64 which is subject to the memory consumptions.

*J*=1*N* ∑ *i* (*W* 1 ∗*x* *i* −*y* *i* ) 2  J=1N∑i(W1∗xi−yi)2

If the cost is very small, we may use the total cost rather than the average cost to make a more precise in the floating point math for the derivative.

Back propagation

**Back propagate your loss to adjust W.** To compute the partial derivatives, ∂*J*∂*W* *i*   ∂J∂Wi , we can start from each node in the left most layer and propagate the gradient until it reaches the rightmost layer. Then, we move to the next layer and start the process again. For a deep network, this is very inefficient. To compute the partial gradient efficiently, we perform a forward pass and a back propagation.

Forward pass

First, we compute the cost of a forward pass:

Keep track of the naming of your input and output, its **shape** (dimension) and the equations. This is one great tip when you program DL. (N,) means a 1-D array with N elements. (N,1) means a 2-D array with N rows each containing 1 element. (N, 3, 4) means a 3-D array.

This method “forward” computes the equation below:

*out*=*W* 1 ∗*X* 1 +*W* 2 ∗*X* 2 +*b* out=W1∗X1+W2∗X2+b

**def** **forward**(x, W, b):

*# x: input sample (N, 2)*

*# W: Weight (2,)*

*# b: bias float*

*# out: (N,)*

out **=** x**.**dot(W) **+** b *# Multiple X with W + b: (N, 2) \* (2,) -> (N,)*

**return** out

To compute the mean square loss:

*J*=1*N* ∑ *i* (*out*−*y* *i* ) 2  J=1N∑i(out−yi)2

**def** **mean\_square\_loss**(h, y):

*# h: prediction (N,)*

*# y: true value (N,)*

N **=** X**.**shape[0] *# Find the number of samples*

loss **=** np**.**sum(np**.**square(h **-** y)) **/** N *# Compute the mean square error from its true value y*

**return** loss

Back propagation pass

**Find the derivative by back propagation**. We back propagate the gradient from the right most layer to the left in one single pass.

Compute the first partial derivative ∂*J*∂*out* *i*   ∂J∂outi from the right most layer.

*J*=1*N* ∑ *i* (*out* *i* −*y* *i* ) 2  J=1N∑i(outi−yi)2

*J* *i* =1*N* (*out* *i* −*y* *i* ) 2  Ji=1N(outi−yi)2

∂*f*∂*out* *i*  =2*N* (*out* *i* −*y* *i* ) ∂f∂outi=2N(outi−yi)

We add a line of code below in the mean square loss to compute ∂*J*∂*out* *i*   ∂J∂outi

**def** **mean\_square\_loss**(h, y):

*# h: prediction (N,)*

*# y: true value (N,)*

**...**

dout **=** 2 **\*** (h**-**y) **/** N *# Compute the partial derivative of J relative to out*

**return** loss, dout

In DL programing, we often name our back propagation derivative as:

∂ J∂ out  as dout ∂ J∂ out as dout

∂ f∂ var  as dvar ∂ f∂ var as dvar

**Use the chain rule to back propagate the gradient.** Now we have ∂*J*∂*out* *i*   ∂J∂outi . We apply the simple chain rule in calculus to back propagate the gradient one more layer to the left. We follow our naming convention to name ∂*J*∂*W*  as *dW* ∂J∂W as dW and ∂*J*∂*b*  as *db* ∂J∂b as db .

Apply chain rule:

*J*=1*N* ∑ *i* (*out* *i* −*y* *i* ) 2  J=1N∑i(outi−yi)2

∂*J*∂*W* =∂*J*∂*out* ∂*out*∂*W*  ∂J∂W=∂J∂out∂out∂W

∂*J*∂*b* =∂*J*∂*out* ∂*out*∂*b*  ∂J∂b=∂J∂out∂out∂b

With the equation *out* out , we take the partial derivative.

*out*=*W*∗*X*+*b* out=W∗X+b

∂*out*∂*W* =*X* ∂out∂W=X

∂*out*∂*b* =1 ∂out∂b=1

Apply the derivative to the chain rule:

∂*J*∂*W* =∂*J*∂*out* ∂*out*∂*W* =∂*J*∂*out* *X* ∂J∂W=∂J∂out∂out∂W=∂J∂outX

∂*J*∂*b* =∂*J*∂*out*  ∂J∂b=∂J∂out

A lot of mathematical notation is involved, but the code for *dW*,*db* dW,db is pretty simple.

**def** **forward**(x, W, b):

out **=** x**.**dot(W) **+** b

cache **=** (x, W, b)

**return** out, cache

**def** **backward**(dout, cache):

*# dout: dJ/dout (N,)*

*# x: input sample (N, 2)*

*# W: Weight (2,)*

*# b: bias float*

x, W, b **=** cache

dW **=** x**.**T**.**dot(dout) *# Transpose x (N, 2) -> (2, N) and multiple with dout. (2, N) \* (N, ) -> (2,)*

db **=** np**.**sum(dout, axis**=**0) *# Add all dout (N,) -> scalar*

**return** dW, db

**def** **compute\_loss**(X, W, b, y**=**None):

h, cache **=** forward(X, W, b)

loss, dout **=** mean\_square\_loss(h, y)

dW, db **=** backward(dout, cache)

**return** loss, dW, db

Here is the full listing of the code:

import numpy as np

iteration **=** 10000

learning\_rate **=** 1e-6

N **=** 100

**def** **true\_y**(education, income):

""" Find the number of dates corresponding to the years of education and monthly income.

Instead of collecting the real data, an Oracle provides us a formula to compute the true value.

"""

dates **=** 0.8 **\*** education **+** 0.003 **\*** income **+** 2

**return** dates

**def** **sample**(education, income):

"""We generate sample of possible dates from education and income value.

Instead of collecting the real data, we find the value from the true\_y

and add some noise to make it looks like sample data.

"""

dates **=** true\_y(education, income)

dates **+=** dates **\*** 0.01 **\*** np**.**random**.**randn(education**.**shape[0]) *# Add some noise*

**return** dates

**def** **forward**(x, W, b):

*# x: input sample (N, 2)*

*# W: Weight (2,)*

*# b: bias float*

*# out: (N,)*

out **=** x**.**dot(W) **+** b *# Multiple X with W + b: (N, 2) \* (2,) -> (N,)*

cache **=** (x, W, b)

**return** out, cache

**def** **mean\_square\_loss**(h, y):

*# h: prediction (N,)*

*# y: true value (N,)*

N **=** X**.**shape[0] *# Find the number of samples*

loss **=** np**.**sum(np**.**square(h **-** y)) **/** N *# Compute the mean square error from its true value y*

dout **=** 2 **\*** (h**-**y) **/** N *# Compute the partial derivative of J relative to out*

**return** loss, dout

**def** **backward**(dout, cache):

*# dout: dJ/dout (N,)*

*# x: input sample (N, 2)*

*# W: Weight (2,)*

*# b: bias float*

x, W, b **=** cache

dw **=** x**.**T**.**dot(dout) *# Transpose x (N, 2) -> (2, N) and multiple with dout. (2, N) \* (N, ) -> (2,)*

db **=** np**.**sum(dout, axis**=**0) *# Add all dout (N,) -> scalar*

**return** dw, db

**def** **compute\_loss**(X, W, b, y**=**None):

h, cache **=** forward(X, W, b)

**if** y **is** None:

**return** h

loss, dout **=** mean\_square\_loss(h, y)

dW, db **=** backward(dout, cache)

**return** loss, dW, db

education **=** np**.**random**.**randint(26, size**=**N) *# (N,) Generate 10 random sample with years of education from 0 to 25.*

income **=** np**.**random**.**randint(10000, size**=**education**.**shape[0]) *# (N,) Generate the corresponding income.*

*# The number of dates according to the formula of an Oracle.*

*# In practice, the value come with each sample data.*

Y **=** sample(education, income) *# (N,)*

W **=** np**.**array([0.1, 0.1]) *# (2,)*

b **=** 0

X **=** np**.**concatenate((education[:, np**.**newaxis], income[:, np**.**newaxis]), axis**=**1) *# (N, 2) N samples with 2 features*

**for** i **in** range(iteration):

loss, dW, db **=** compute\_loss(X, W, b, Y)

W **-=** learning\_rate **\*** dW

b **-=** learning\_rate **\*** db

**if** i**%**100**==**0:

**print**(f"iteration {i}: loss={loss:.4} W1={W[0]:.4} dW1={dW[0]:.4} W2={W[1]:.4} dW2={dW[1]:.4} b= {b:.4} db = {db:.4}")

**print**(f"W = {W}")

**print**(f"b = {b}")

General principle in back propagation

The machine learning library provides pre-built layers with feed forward and back propagation. Many DL class assignments spend a large amount of time in back propagation. With vectorization and some ad hoc functions, the process is error prone, but not necessarily difficult. Let’s summarize the above steps above with some good tips.

Draw the forward pass and back propagation pass with clear notification of variables that we used in the program. Add functions, derivatives and the shape for easy reference.

Perform a forward pass to calculate the output and the cost:

*out*=*W* 1 ⋅*X* 1 +*W* 2 ⋅*X* 2 +*b* out=W1⋅X1+W2⋅X2+b

*J*=1*N* ∑ *i* (*out*−*y* *i* ) 2  J=1N∑i(out−yi)2

Find the partial derivative of the cost:

∂*J*∂*out* =2*N* (*out*−*y*) ∂J∂out=2N(out−y)

For every node, find the derivative of the function:

*out*=*W*∗*X*+*b* out=W∗X+b

∂*out*∂*W* =*X* ∂out∂W=X

∂*out*∂*b* =1 ∂out∂b=1

Find the total gradient with the chain rule from right to left:

∂*J*∂*l* *k*−1  =∂*J*∂*l* *k*  ∂*l* *k* ∂*l* *k*−1   ∂J∂lk−1=∂J∂lk∂lk∂lk−1

dl1≡∂*J*∂*l* 1  =∂*J*∂*out* ∂*out*∂*l* 1  =∂*J*∂*out* ∂*f* 2 ∂*l* 1   dl1≡∂J∂l1=∂J∂out∂out∂l1=∂J∂out∂f2∂l1

dl1=dout⋅∂*f* 2 ∂*l* 1   dl1=dout⋅∂f2∂l1

Similary,

dW=dl1⋅∂*f* 1 ∂*W*  dW=dl1⋅∂f1∂W

**Know the shape (dimension) of your variables**. Put the function derivative in the diagram. Always make the shape (dimension) of the variables clear in the diagram and in the code. Use this information to build and verify your math operations. For example, multiplying a (N, C, D) matrix with a (D, K) matrix should produce a (N, C, K) matrix. Vectorization may confuse you. Expand the equation with sub-indexes with only a couple of weights and features in order to work through the math.

More on back propagation

In back propagation, we may back propagate multiple paths back to the same node. To compute the gradient correctly, we need to add both paths together:

∂*J*∂*o* 3  =∂*J*∂*o* 4  ∂*f* 4 ∂*o* 3  +∂*J*∂*o* 5  ∂*f* 4 ∂*o* 3   ∂J∂o3=∂J∂o4∂f4∂o3+∂J∂o5∂f4∂o3

do3=do4∂*f* 4 ∂*o* 3  +do5∂*f* 4 ∂*o* 3   do3=do4∂f4∂o3+do5∂f4∂o3

Testing the model

I strongly recommend that you think about a linear regression problem that interests you and train a simple model. A lot of issues occur in a complex model and will also show up in a simple model. With a complex model, you treat it as a black box.Many actions are purely random guesses. Working with a model designed by yourself, you create better scenarios to test your theories and develop a better insight in DL. Most tutorials have pre-cooked parameters. They teach you the easier part without having you struggle on the hard part.

So, let Pieter train the system.

iteration 0: loss=2.825e+05 W1=0.09882 dW1=1.183e+04 W2=-0.4929 dW2=5.929e+06 b= -8.915e-05 db = 891.5

iteration 200: loss=3.899e+292 W1=-3.741e+140 dW1=4.458e+147 W2=-1.849e+143 dW2=2.203e+150 b= -2.8e+139 db = 3.337e+146

iteration 400: loss=inf W1=-1.39e+284 dW1=1.656e+291 W2=-6.869e+286 dW2=8.184e+293 b= -1.04e+283 db = 1.24e+290

iteration 600: loss=nan W1=nan dW1=nan W2=nan dW2=nan b= nan db = nan

The application overflows within 600 iterations! Since the loss and the gradient are so high, we test whether the learning rate is too high. We decrease the learning rate to 1e-8.We do not have the overflow problem, but the loss remains high.

iteration 90000: loss=4.3e+01 W1= 0.23 dW1=-1.3e+02 W2=0.0044 dW2= 0.25 b= 0.0045 db = -4.633

W = [ 0.2437896 0.00434705]

b = 0.004981262980767952

We are reluctant to take actions without information. However, since the application runs fast, we will give it one more try . With 10,000,000 iterations and a learning rate of 1e-10, the loss remains very high. It will be better to trace the source of problem now.

iteration 9990000: loss=3.7e+01 W1= 0.22 dW1=-1.1e+02 W2=0.0043 dW2= 0.19 b= 0.0049 db = -4.593

W = [ 0.22137005 0.00429005]

b = 0.004940551119084607

Tracing gradient is a powerful tool in DL debugging.

Even the loss shows similar symptoms such as a bad learning rate, We suspect this is not the root cause. After some tracing, we realize the gradient is very high. We plot the cost function relative to *W* W to illustrate the real issue.

This is a U shape curve which is different from a bowl shape curve that we used in the gradient descent explanation.

The y-axis is *W* 2  W2 (monthly income) and the x-axis is *W* 1  W1 (years of education). Cost responses are more aggressive with *W* 2  W2 than *W* 1  W1 . Monthly income ranges from 0 to 10,000 and years of education range from 0 to 30. Obviously, the different scale in these two features causes a major difference in its gradient. Because of the different scale, we cannot have a single learning rate that works well for both of them. The solution is pretty simple with a couple line of code changes. We re-scale the income value.

**def** **true\_y**(education, income):

dates **=** 0.8 **\*** education **+** 0.3 **\*** income **+** 2

**return** dates

**...**

income **=** np**.**random**.**randint(10, size**=**education**.**shape[0]) *# (N,) Generate the corresponding income.*

Here is the output which is close to our true model:

iteration 0: loss=518.7 W1=0.1624 dW1=-624.5 W2=0.3585 dW2=-2.585e+03 b= 0.004237 db = -42.37

iteration 10000: loss=0.4414 W1=0.8392 dW1=0.0129 W2=0.3128 dW2=0.004501 b= 0.5781 db = -0.4719

iteration 20000: loss=0.2764 W1=0.8281 dW1=0.009391 W2=0.3089 dW2=0.003277 b= 0.9825 db = -0.3436

iteration 30000: loss=0.1889 W1=0.8201 dW1=0.006837 W2=0.3061 dW2=0.002386 b= 1.277 db = -0.2501

iteration 40000: loss=0.1425 W1=0.8142 dW1=0.004978 W2=0.3041 dW2=0.001737 b= 1.491 db = -0.1821

iteration 50000: loss=0.1179 W1=0.81 dW1=0.003624 W2=0.3026 dW2=0.001265 b= 1.647 db = -0.1326

iteration 60000: loss=0.1049 W1=0.8069 dW1=0.002639 W2=0.3015 dW2=0.0009208 b= 1.761 db = -0.09653

iteration 70000: loss=0.09801 W1=0.8046 dW1=0.001921 W2=0.3007 dW2=0.0006704 b= 1.844 db = -0.07028

iteration 80000: loss=0.09435 W1=0.803 dW1=0.001399 W2=0.3001 dW2=0.0004881 b= 1.904 db = -0.05117

iteration 90000: loss=0.09241 W1=0.8018 dW1=0.001018 W2=0.2997 dW2=0.0003553 b= 1.948 db = -0.03725

W = [ 0.80088848 0.29941657]

b = 1.9795590414763997

Non-linearity

Pieter comes back and complains our linear model is not adequate. Pieter claims the relationship between the years of education and the number of dates are not exactly linear. We should give more rewards for people holding advance degrees.

Can we combine 2 linear functions with multiple layers to form a non-linear function?

*f*(*x*)=*Wx*+*b* f(x)=Wx+b

*g*(*z*)=*Uz*+*c* g(z)=Uz+c

The answer is no.

*g*(*f*(*x*))=*U*(*Wx*+*b*)+*c*=*Vx*+*d* g(f(x))=U(Wx+b)+c=Vx+d

After some thought, we apply the following to the output of a computation node.

*f*(*x*)=*max*(0,*x*) f(x)=max(0,x)

With this function, in theory, we can construct the non-linear relations that Pieter wants.

Adding both output:

Adding a non-linear function after a linear equation enriches the complexity of our model. These methods are called **activation functions**. Common functions are tanh and ReLU.

Sigmoid

Sigmoid is one of the earliest functions used in deep networks. Nevertheless, as an activation function, its importance has gradually been replaced by other functions like ReLU. Currently, the sigmoid function is more popular as a gating function in LSTM/GRU (an “on/off” gate) to selectively remember or forget information. The discussion of a sigmoid function often acts as a showcase of explaining issues with deep networks.

ReLU

ReLU is one of the most popular activation functions. Its popularity arrives because it works better with gradient descent. It performs better than the sigmoid function because the sigmoid node saturates easily and works less efficiently with gradient descent. (The reasons why some functions work better in gradient descent will be explained later.)

*y*=*max*(0,*x*) y=max(0,x)

tanh

tanh is similar to sigmoid but the output is within [-1, 1] instead of [0, 1].

As mention before, we want the gradient descent to follow the curve of the cost function but not in some zip zap pattern. For the sigmoid function, the output value is always positive (between 0 and 1). According to the formular below, the sign for the gradient for *W* W subjects to the sign of ∂*J*∂*l* *k*+1   ∂J∂lk+1 because *X* X is always positive in a sigmoid layer output. So, the derivatives for this layer are either all positive or all negative. Instead of following the curve, all *W* W in this layer move in the same direction in a zip zap pattern.

∂*l* *k* ∂*W* =*X*⋅∂*J*∂*l* *k*+1   ∂lk∂W=X⋅∂J∂lk+1

Since tanh has both positive and negative outputs, tanh does not have this problem as sigmoid.

Numpy provides all API to implement these functions.

import numpy as np

import matplotlib.pyplot as plt

**def** **sigmoid**(x):

**return** 1 **/** (1 **+** np**.**exp(**-**x))

**def** **relu**(x):

**return** np**.**maximum(0, x)

**def** **tanh**(x):

**return** np**.**tanh(x)

x **=** np**.**arange(**-**20, 20, 0.001)

y **=** sigmoid(x)

plt**.**axvline(x**=**0, color**=**"0.8")

plt**.**plot(x, y)

plt**.**show()

y **=** relu(x)

plt**.**axvline(x**=**0, color**=**"0.8")

plt**.**axhline(y**=**0, color**=**"0.8")

plt**.**plot(x, y)

plt**.**show()

y **=** tanh(x)

plt**.**plot(x, y)

plt**.**axhline(y**=**0, color**=**"0.8")

plt**.**show()

Feed forward and back propagation with sigmoid or ReLU

The derivative for ReLU function:

*f*(*x*)=max(0,*x*) f(x)=max(0,x)

*f* ′ (*x*)={1,0, if *x*>0otherwise  f′(x)={1,if x>00,otherwise

The derivative for sigmoid:

*σ*(*x*)=11+*e* −*x*   σ(x)=11+e−x

With some calculus:

∂*σ*(*x*)∂*x* =1(1+*e* −*x* ) 2  ⋅*e* −*x* =1(1+*e* −*x* ) ⋅1(1+*e* *x* )  ∂σ(x)∂x=1(1+e−x)2⋅e−x=1(1+e−x)⋅1(1+ex)

∂*σ*(*x*)∂*x* =1(1+*e* −*x* ) ⋅(1−11+*e* −*x*  ) ∂σ(x)∂x=1(1+e−x)⋅(1−11+e−x)

∂*σ*(*x*)∂*x* =*σ*(*x*)⋅(1−*σ*(*x*)) ∂σ(x)∂x=σ(x)⋅(1−σ(x))

**def** **relu\_forward**(x):

cache **=** x

out **=** np**.**maximum(0, x)

**return** out, cache

**def** **relu\_backward**(dout, cache):

out **=** cache

dh **=** dout

dh[out **<** 0] **=** 0

**return** dh

**def** **sigmoid\_forward**(x):

out **=** 1 **/** (1 **+** np**.**exp(**-**x))

**return** out, out

**def** **sigmoid\_backward**(dout, cache):

dh **=** cache **\*** (1**-**cache)

**return** dh **\*** dout

Fully connected network

Let’s apply all our knowledge to build a fully connected network:

For each node in the hidden layer (except the last layer), we apply:

*z* *j* =∑ *i* *W* *ij* *x* *i* +*b* *i*  zj=∑iWijxi+bi

*h*(*z* *j* )=max(0,*z* *j* ) h(zj)=max(0,zj)

For the layer before the output, we apply only the linear equation but not the ReLU.

Most of the code is already done in previous examples. We add the forward pass code for 4 hidden layers.

z1, cache\_z1 **=** affine\_forward(X, W[0], b[0])

h1, cache\_relu1 **=** relu\_forward(z1)

z2, cache\_z2 **=** affine\_forward(h1, W[1], b[1])

h2, cache\_relu2 **=** relu\_forward(z2)

z3, cache\_z3 **=** affine\_forward(h2, W[2], b[2])

h3, cache\_relu3 **=** relu\_forward(z3)

z4, cache\_z4 **=** affine\_forward(h3, W[3], b[3])

Here is the corresponding back propagation:

dz4, dW[3], db[3] **=** affine\_backward(dout, cache\_z4)

dh3 **=** relu\_backward(dz4, cache\_relu3)

dz3, dW[2], db[2] **=** affine\_backward(dh3, cache\_z3)

dh2 **=** relu\_backward(dz3, cache\_relu2)

dz2, dW[1], db[1] **=** affine\_backward(dh2, cache\_z2)

dh1 **=** relu\_backward(dz2, cache\_relu1)

\_, dW[0], db[0] **=** affine\_backward(dh1, cache\_z1)

For those interested in details, we relist some of the code.

iteration **=** 100000

learning\_rate **=** 1e-4

N **=** 100

**...**

**def** **affine\_forward**(x, W, b):

*# x: input sample (N, D)*

*# W: Weight (2, k)*

*# b: bias (k,)*

*# out: (N, k)*

out **=** x**.**dot(W) **+** b *# (N, D) \* (D, k) -> (N, k)*

cache **=** (x, W, b)

**return** out, cache

**def** **affine\_backward**(dout, cache):

*# dout: dJ/dout (N, K)*

*# x: input sample (N, D)*

*# W: Weight (D, K)*

*# b: bias (K, )*

x, W, b **=** cache

**if** len(dout**.**shape)**==**1:

dout **=** dout[:, np**.**newaxis]

**if** len(W**.**shape) **==** 1:

W **=** W[:, np**.**newaxis]

W **=** W**.**T

dx **=** dout**.**dot(W)

dW **=** x**.**T**.**dot(dout)

db **=** np**.**sum(dout, axis**=**0)

**return** dx, dW, db

**def** **relu\_forward**(x):

cache **=** x

out **=** np**.**maximum(0, x)

**return** out, cache

**def** **relu\_backward**(dout, cache):

out **=** cache

dh **=** dout

dh[out **<** 0] **=** 0

**return** dh

**def** **mean\_square\_loss**(h, y):

*# h: prediction (N,)*

*# y: true value (N,)*

N **=** X**.**shape[0] *# Find the number of samples*

h **=** h**.**reshape(**-**1)

loss **=** np**.**sum(np**.**square(h **-** y)) **/** N *# Compute the mean square error from its true value y*

nh **=** h**.**reshape(**-**1)

dout **=** 2 **\*** (nh**-**y) **/** N *# Compute the partial derivative of J relative to out*

**return** loss, dout

**def** **compute\_loss**(X, W, b, y**=**None, use\_relu**=**True):

z1, cache\_z1 **=** affine\_forward(X, W[0], b[0])

h1, cache\_relu1 **=** relu\_forward(z1)

z2, cache\_z2 **=** affine\_forward(h1, W[1], b[1])

h2, cache\_relu2 **=** relu\_forward(z2)

z3, cache\_z3 **=** affine\_forward(h2, W[2], b[2])

h3, cache\_relu3 **=** relu\_forward(z3)

z4, cache\_z4 **=** affine\_forward(h3, W[3], b[3])

**if** y **is** None:

**return** z4, None, None

dW **=** [None] **\*** 4

db **=** [None] **\*** 4

loss, dout **=** mean\_square\_loss(z4, y)

dz4, dW[3], db[3] **=** affine\_backward(dout, cache\_z4)

dh3 **=** relu\_backward(dz4, cache\_relu3)

dz3, dW[2], db[2] **=** affine\_backward(dh3, cache\_z3)

dh2 **=** relu\_backward(dz3, cache\_relu2)

dz2, dW[1], db[1] **=** affine\_backward(dh2, cache\_z2)

dh1 **=** relu\_backward(dz2, cache\_relu1)

\_, dW[0], db[0] **=** affine\_backward(dh1, cache\_z1)

**return** loss, dW, db

**...**

W **=** [None] **\*** 4

b **=** [None] **\*** 4

W[0] **=** np**.**array([[0.7, 0.05, 0.0, 0.2, 0.2], [0.3, 0.01, 0.01, 0.3, 0.2]]) *# (2, K)*

b[0] **=** np**.**array([0.8, 0.2, 1.0, 0.2, 0.1])

W[1] **=** np**.**array([[0.7, 0.05, 0.0, 0.2, 0.1], [0.3, 0.01, 0.01, 0.3, 0.1], [0.3, 0.01, 0.01, 0.3, 0.1], [0.07, 0.04, 0.01, 0.1, 0.1], [0.1, 0.1, 0.01, 0.02, 0.03]]) *# (K, K)*

b[1] **=** np**.**array([0.8, 0.2, 1.0, 0.2, 0.1])

W[2] **=** np**.**array([[0.7, 0.05, 0.0, 0.2, 0.1], [0.3, 0.01, 0.01, 0.3, 0.1], [0.3, 0.01, 0.01, 0.3, 0.1], [0.07, 0.04, 0.01, 0.1, 0.1], [0.1, 0.1, 0.01, 0.02, 0.03]]) *# (K, K)*

b[2] **=** np**.**array([0.8, 0.2, 1.0, 0.2, 0.1])

W[3] **=** np**.**array([[0.8], [0.5], [0.05], [0.05], [0.1]]) *# (K, 1)*

b[3] **=** np**.**array([0.2])

X **=** np**.**concatenate((education[:, np**.**newaxis], income[:, np**.**newaxis]), axis**=**1) *# (N, 2) N samples with 2 features*

**for** i **in** range(iteration):

loss, dW, db **=** compute\_loss(X, W, b, Y)

**for** j, (cdW, cdb) **in** enumerate(zip(dW, db)):

W[j] **-=** learning\_rate **\*** cdW

b[j] **-=** learning\_rate **\*** cdb

**if** i**%**20000**==**0:

**print**(f"iteration {i}: loss={loss:.4}")

We also generate some testing data to measure how well our model predicts.

TN **=** 100

test\_education **=** np**.**full(TN, 22)

test\_income **=** np**.**random**.**randint(TN, size**=**test\_education**.**shape[0])

test\_income **=** np**.**sort(test\_income)

true\_model\_Y **=** true\_y(test\_education, test\_income)

true\_sample\_Y **=** sample(test\_education, test\_income, verbose**=**False)

X **=** np**.**concatenate((test\_education[:, np**.**newaxis], test\_income[:, np**.**newaxis]), axis**=**1)

out, \_, \_ **=** compute\_loss(X, W, b)

loss\_model, \_ **=** mean\_square\_loss(out, true\_model\_Y)

loss\_sample, \_ **=** mean\_square\_loss(out, true\_sample\_Y)

**print**(f"testing: loss (compare with Oracle)={loss\_model:.6}")

**print**(f"testing: loss (compare with sample)={loss\_sample:.4}")

We plot the predicted values with the true values. (The values derived from our Oracle model.) For the first plot, we temporarily downsize the network to 2 hidden layers. In the plot, we fix the years of education to 22 and plot how the number of dates varied with income. The orange dots are our predictions. The blue dots are from the Oracle model adding some noise. The data match pretty well with each other.

Congratulations! We just solved a problem using deep learning!

The code looks simple and easy. When we solve couple visual recognition problems later, you will realize that the codes are almost the same. We prefer to keep things simple now so we can play with it using different ideas. For real problems, instead of two inputs (the years of education and the monthly income), there may be a couple dozen features (input). For complex problems, we add fully connected layers (FC) to enrich the model. For visual recognition, we convert the 2-D image into a 1-D array and feed it to an FC. Nevertheless, if we want to push the accuracy up for complex visual problems, we add convolution layers in front of the FC. Nevertheless, learning FC covers the critical techniques for CNN.

Now, we increase the number of hidden layers back from 2 to 4. Our prediction accuracy drops.Iit takes more training time and tuning. When we plot it in 3D with variable income and education, some part of the 2D plain is bent instead of flat.

When we create the training dataset, we add some noise to our true value (# of dates). When we have a complex model, it increases its capability to model the noise also. If we have a large dataset, the effect of noise should cancel out. However, if the training dataset is not big enough and the model is complex, the accuracy suffers in comparison to the simpler model. In this exercise, when we increase the model complexity, it gets more difficult to train and optimizeand the accuracy drops. In general, we start with a simple model and increase its complexity later. It is hard to tell whether we need more tuning/iterations or if it is simply not working if we jump into a complex model too soon.

**def** **sample**(education, income, verbose**=**True):

dates **=** true\_y(education, income)

noise **=** dates **\*** 0.15 **\*** np**.**random**.**randn(education**.**shape[0]) *# Add some noise to our true model.*

**return** dates **+** noise

For completeness, we replace our ReLU function with a sigmoid function and plot the same diagram:

Below is one of the models generated. Because we start with random guess of *W* W , we end up with models with different ||W|| for each run.

Layer 1:

[[ 1.10727659, 0.22189273, 0.13302861, 0.2646622 , 0.2835898 ],

[ 0.23522207, 0.01791731, -0.01386124, 0.28925567, 0.187561 ]]

Layer 2:

[[ 0.9450821 , 0.14869831, 0.07685842, 0.23896402, 0.15320876],

[ 0.33076781, 0.02230716, 0.01925127, 0.30486342, 0.10669098],

[ 0.18483084, -0.03456052, -0.01830806, 0.28216702, 0.07498301],

[ 0.11560201, 0.05810744, 0.021574 , 0.10670155, 0.11009798],

[ 0.17446553, 0.12954657, 0.03042245, 0.03142454, 0.04630127]]),

Layer 3:

[[ 0.79405847, 0.10679984, 0.00465651, 0.20686431, 0.11202472],

[ 0.31141474, 0.01717969, 0.00995529, 0.30057041, 0.10141655],

[ 0.13030365, -0.09887915, 0.0265004 , 0.29536237, 0.07935725],

[ 0.07790114, 0.04409276, 0.01333717, 0.10145275, 0.10112565],

[ 0.12152267, 0.11339623, 0.00993313, 0.02115832, 0.03268988]]),

Layer 4:

[[ 0.67123192],

[ 0.48754364],

[-0.2018187 ],

[-0.03501616],

[ 0.07363663]])]

In the first part of the tutorial, we finish a FC network using a very simple problem. But it allows us to play with it, and learns better insight. Almost the same code and techniques applies to a real problem, so it does not hurt either. Nevertheless, training a deep network is not simple. In the second part of the tutorial, we cover the critical DL issues and its resolutions. We will also put things together with 2 visual recognition problems.

[Part 2 of the deep learning can be found here.](https://jhui.github.io/2017/03/17/Deep-learning-tutorial-2/)

Please enable JavaScript to view the <a href="http://disqus.com/?ref\_noscript">comments powered by Disqus.</a>

* Jonathan Hui blog
* [jhui](https://github.com/jhui)

“Software architect”