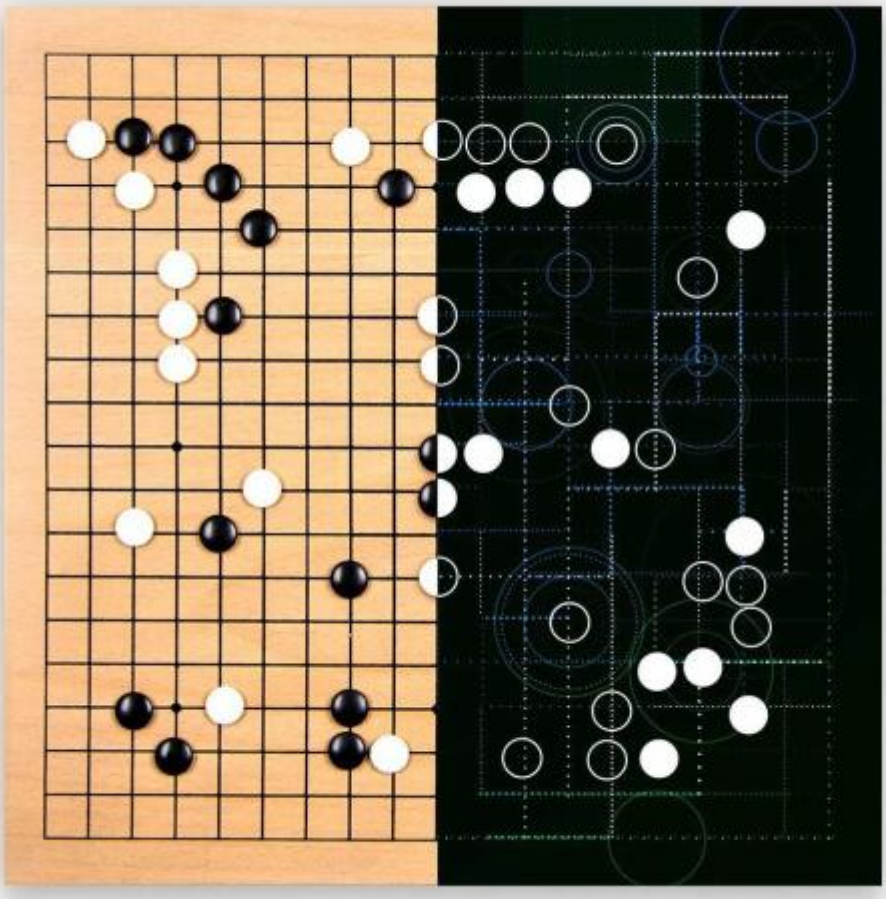
从代码中追溯深度学习的历史

深度学习发展到如今的地位，离不开下面这 6 段代码。本文介绍了这些代码的创作者及其完成这些突破性成就的故事背景。每个故事都有简单的代码示例，读者们可以在 FloydHub 和 GitHub 找到相关代码。



Source: Google press image 图片来源：Google 新闻图片

要运行 FloydHub 上的代码示例，请确保您的电脑已经安装了 Floyd 命令行工具，并将我上传的代码示例克隆到本地计算机。如果您是第一次使用 FloydHub，可以先阅读我之前发布的文章中关于如何使用 FloydHub 的*部分*。在本地计算机示例项目中初始化命令行界面之后，您就可以运行以下命令在 FloydHub 上启动项目：

```
floyd run --data emilwallner/datasets/mnist/1:mnist --tensorboard --mode jupyter
```

最小二乘法

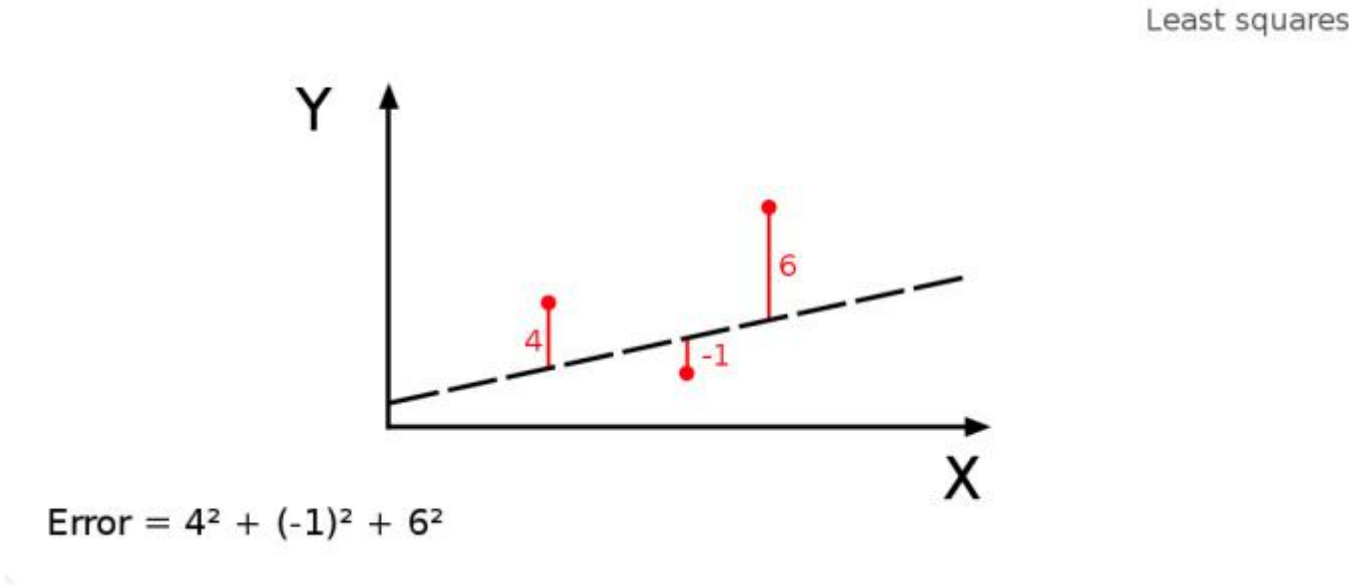
所有的深度学习算法都始于下面这个数学公式（我已将其转成 Python 代码）

```
Python
# y = mx + b
# m is slope, b is y-intercept

def compute_error_for_line_given_points(b, m, coordinates):
    totalError = 0
    for i in range(0, len(coordinates)):
        x = coordinates[i][0]
        y = coordinates[i][1]
        totalError += (y - (m * x + b)) ** 2
    return totalError / float(len(coordinates))
# example
compute_error_for_line_given_points(1, 2, [[3,6],[6,9],[12,18]])
```

最小二乘法在 1805 年由 Adrien-Marie Legendre 首次提出（1805, Legendre），这位巴黎数学家也以测量仪器闻名。他极其痴迷于预测彗星的方位，坚持不懈地寻找一种可以基于彗星方位历史数据计算其轨迹的算法。

他尝试了许多种算法，一遍遍试错，终于找到了一个算法与结果相符。Legendre 的算法是首先预测彗星未来的方位，然后计算误差的平方，最终目的是通过修改预测值以减少误差平方和。而这也正是线性回归的基本思想。

读者可以在 Jupyter notebook 中运行上述代码来加深对这个算法的理解。m 是系数，b 是预测的常数项，coordinates 是彗星的位置。目标是找到合适的 m 和 b 使其误差尽可能小。

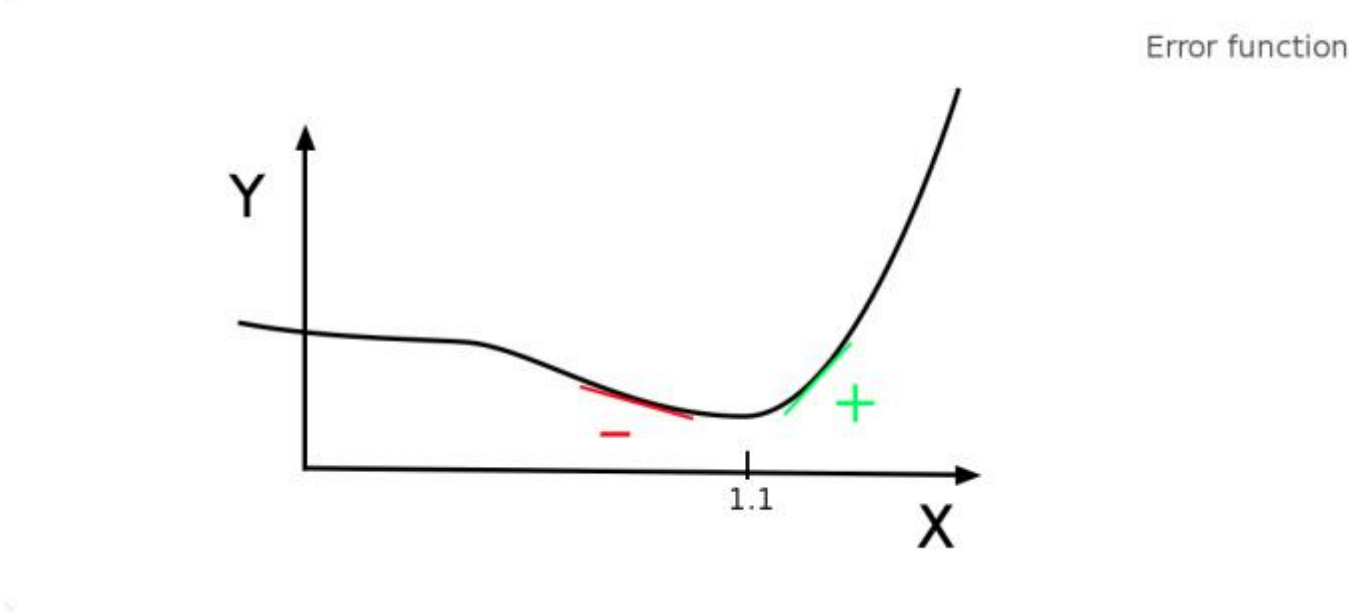


这是深度学习的核心思想：给定输入值和期望的输出值，然后寻找两者之间的相关性。

梯度下降

Legendre 这种通过手动尝试来降低错误率的方法非常耗时。荷兰的诺贝尔奖得主 Peter Debye 在一个世纪后（1909 年）正式提出了一种简化这个过程的方法。

假设 Legendre 的算法需要考虑一个参数 —— 我们称之为 X 。Y 轴表示每个 X 的误差值。Legendre 的算法是找到使得误差最小的 X。在下图中，我们可以看到当 X = 1.1 时，误差 Y 取到最小值。



Peter Debye 注意到最低点左边的斜率是负的，而另一边则是正的。因此，如果知道了任意给定 X 的斜率值，就可以找到 Y 的最小值点。

这便是梯度下降算法的基本思想。几乎所有的深度学习模型都会用到梯度下降算法。

要实现这个算法，我们假设误差函数是 Error = x ^ 5 -2x ^ 3-2。要得到任意给定 X 的斜率，我们需要对其求导，即 5x^4 – 6x^2：

如果您需要复习导数的相关知识，请观看 Khan Academy 的视频。

下面我们用 Python 实现 Debye 的算法：

```python
current_x = 0.5 # the algorithm starts at x=0.5
learning_rate = 0.01 # step size multiplier
num_iterations = 60 # the number of times to train the function

#the derivative of the error function (x**4 = the power of 4 or x^4)
def slope_at_given_x_value(x):
   return 5 * x**4 - 6 * x**2
```

```
8
9# Move X to the right or left depending on the slope of the error function
10for i in range(num_iterations):
11    previous_x = current_x
12    current_x += -learning_rate * slope_at_given_x_value(previous_x)
13    print(previous_x)
14
15print("The local minimum occurs at %f" % current_x)
```

这里的窍门在于 learning_rate。我们通过沿斜率的相反方向行进来逼近最低点。此外，越接近最低点，斜率越小。因此当斜率接近零时，每一步下降的幅度会越来越小。

num_iterations 是你预计到达最小值之前所需的迭代次数。可以通过调试该参数训练自己关于梯度下降算法的直觉。

线性回归

最小二乘法配合梯度下降算法，就是一个完整的线性回归过程。在 20 世纪 50 年代和 60 年代，一批实验经济学家在早期的计算机上实现了这些想法。这个过程是通过实体打卡 —— 真正的手工软件程序实现的。准备这些打孔卡就需要几天的时间，而通过计算机进行一次回归分析最多需要 24 小时。

下面是用 Python 实现线性回归的一个示例（我们不需要在打卡机上完成这个操作）：

Python

```
1#Price of wheat/kg and the average price of bread
2wheat_and_bread = [[0.5,5],[0.6,5.5],[0.8,6],[1.1,6.8],[1.4,7]]
3
4def step_gradient(b_current, m_current, points, learningRate):
5    b_gradient = 0
6    m_gradient = 0
7    N = float(len(points))
8    for i in range(0, len(points)):
9        x = points[i][0]
10       y = points[i][1]
```

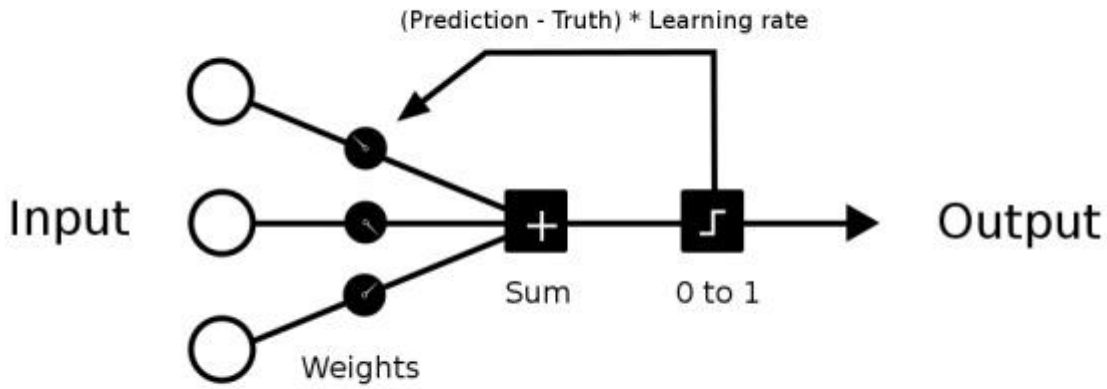
```
        b_gradient += -(2/N) * (y - ((m_current * x) + b_current))
        m_gradient += -(2/N) * x * (y - ((m_current * x) + b_current))
    new_b = b_current - (learningRate * b_gradient)
    new_m = m_current - (learningRate * m_gradient)
    return [new_b, new_m]

def gradient_descent_runner(points, starting_b, starting_m,
learning_rate, num_iterations):
    b = starting_b
    m = starting_m
    for i in range(num_iterations):
        b, m = step_gradient(b, m, points, learning_rate)
    return [b, m]

gradient_descent_runner(wheat_and_bread, 1, 1, 0.01, 100)
```

线性回归本身并没有引入什么新的内容。但是，如何将梯度下降算法运用到误差函数上就需要动动脑子了。运行代码并使用这个线性回归模拟器来加深你的理解吧。

感知机

接下来让我们来认识一下 Frank Rosenblatt。这是一个白天解剖老鼠大脑，晚上寻找外星生命迹象的家伙。1958 年，他发明了一个模仿神经元的机器（1958, Rosenblatt），并因此登上《纽约时报》的头条：“New Navy Device Learns By Doing”。

如果向 Rosenblatt 的机器展示 50 组分别在左右两侧有标记的图像，它可以在没有预先编程的情况下分辨出两张图像（标记的位置）。大众被这个可能真正拥有学习能力的机器震惊了。



The Perceptron

如上图所示，每个训练周期都是从左侧输入数据开始。给所有输入数据添加一个初始的随机权重。然后将它们相加。如果总和为负，将其输出为 0，否则输出为 1。

如果预测结果是正确的，就不改变循环中的权重。如果预测结果是错误的，可以用误差乘以学习率来相应地调整权重。

我们用经典的“或”逻辑来运行感知机。

输入		输出
0	0 =	0
0	1 =	1
1	0 =	1
1	1 =	1

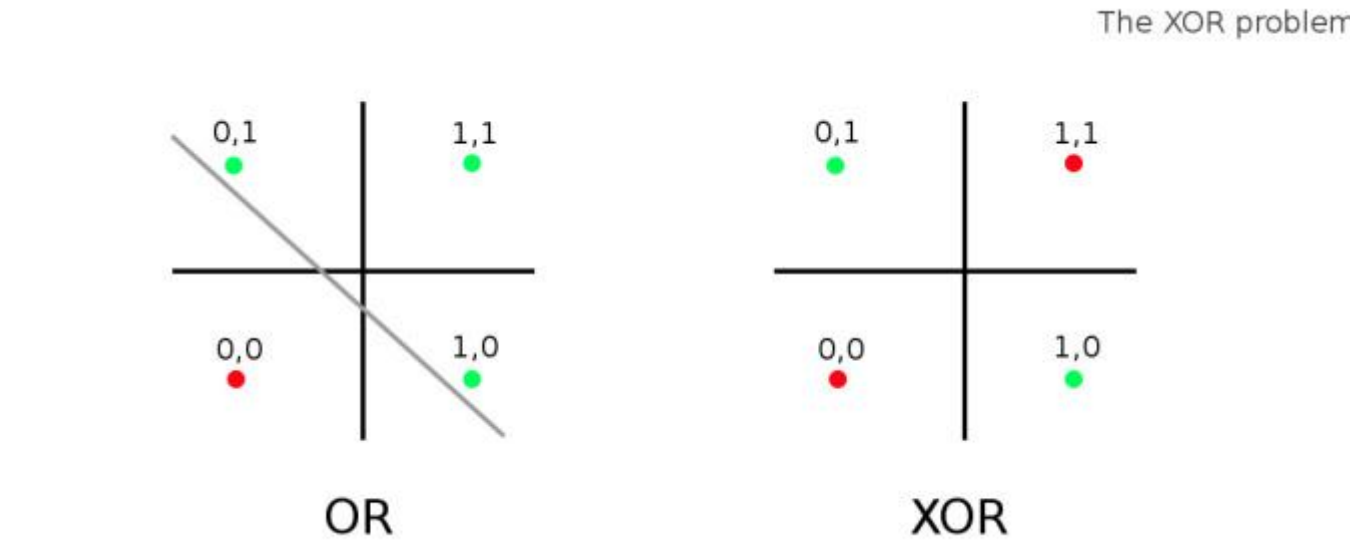
下面是用 Python 实现的感知机模型：

```python
from random import choice
from numpy import array, dot, random
1_or_0 = lambda x: 0 if x < 0 else 1
training_data = [ (array([0,0,1]), 0),
                  (array([0,1,1]), 1),
                  (array([1,0,1]), 1),
                  (array([1,1,1]), 1), ]
weights = random.rand(3)
errors = []
learning_rate = 0.2
num_iterations = 100

for i in range(num_iterations):
    input, truth = choice(training_data)
    result = dot(weights, input)
    error = truth - 1_or_0(result)
    errors.append(error)
    weights += learning_rate * error * input

for x, _ in training_data:
    result = dot(x, w)
    print("{}: {} -> {}".format(input[:2], result, 1_or_0(result)))
```

经过最初的炒作一年之后，Marvin Minsky 和 Seymour Papert 击碎了这个想法（1969, Minsky & Papert）。当时，Minsky 和 Papert 都在麻省理工学院的 AI 实验室工作。他们写了一本书，证明感知机只能解决线性问题。他们还批判了关于多层感知机的想法。可悲的是，Frank Rosenblatt 两年后因船难去世。

在 Minsky 和 Papert 的书籍出版一年之后，一位芬兰硕士研究生提出了用多层感知机解决非线性问题的理论（Linnainmaa, 1970）。由于业内主流对感知机普遍不看好，十多年来 AI 的研究资金也非常短缺。这是 AI 首次遇冷。

Minsky 和 Papert 对感知机的批判主要针对“异或”问题。这个逻辑与“或”逻辑相同，但有一个例外 —— 对两个 true 语句取和（1&1）时，结果返回 False（0）。



如上图所示，在“或”逻辑中，我们可以将 true 和 false 分开。但是可以看出，我们无法使用一个线性函数将“异或”逻辑的结果进行区分。

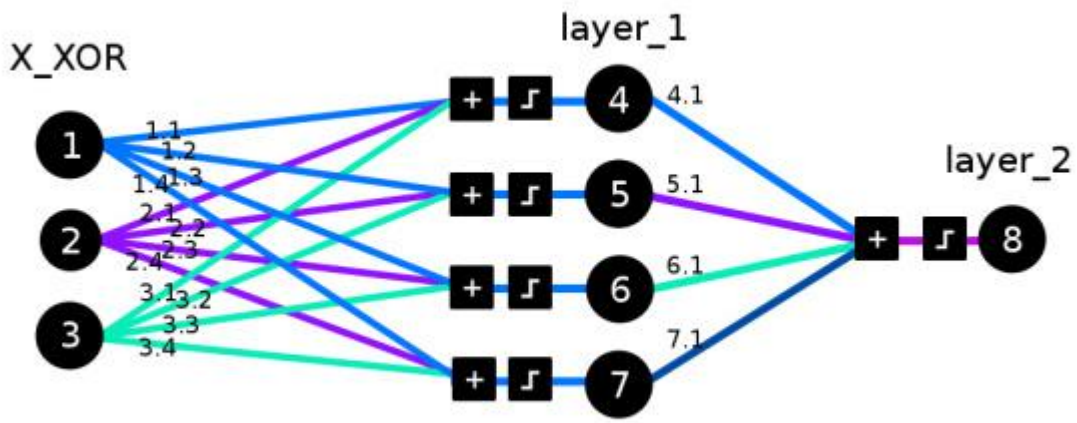
人工神经网络

到 1986 年，几项实验证明，神经网络可以解决复杂的非线性问题（Rumelhart et al., 1986）。 当时计算机的运算速度比该理论提出的时候快了一万倍。Rumelhart 等人是这样介绍他们赫赫有名的论文的：

> 我们描述了一种新的类神经元网络学习过程——反向传播。该过程通过反复调整网络中的连接权重，最小化网络的实际输出向量与期望输出向量之间的差异。调整权重的结果就是，不属于输入或输出的内部“隐藏”单元成为了描述任务域的重要特征，并且这些单元的交互项还捕获了任务中的正则条件。 相较于早期更简单的方法，如“感知机收敛过程”

Nature 323, 533 – 536 (09 October 1986)，反向传播可以创造出有用的新特征。

为了理解这篇文章的核心内容，我会在下面重现 DeepMind 团队 Andrew Trask 的代码。这不是一段普通的代码。它曾被用于斯坦福大学 Andrew Karpathy 的深度学习课程，以及 Siraj Raval 的 Udacity 课程。最重要的是，它解决了“异或”问题，也结束了 AI 遇冷的时代。



Neural Network Map

学习这段代码之前，我们首先通过这个模拟器交互学习一到两个小时来掌握神经网络的核心逻辑。然后阅读 Trask 的博客，然后再阅读四次。需要注意到，X_XOR 数据中添加的参数 [1] 是偏置神经元，它们等价于线性函数中的常数项。

```Python
import numpy as np

X_XOR = np.array([[0,0,1], [0,1,1], [1,0,1],[1,1,1]])
y_truth = np.array([[0],[1],[1],[0]])

np.random.seed(1)
```

```
syn_0 = 2*np.random.random((3,4)) - 1
syn_1 = 2*np.random.random((4,1)) - 1

def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output
def sigmoid_output_to_derivative(output):
    return output*(1-output)

for j in range(60000):
    layer_1 = sigmoid(np.dot(X_XOR, syn_0))
    layer_2 = sigmoid(np.dot(layer_1, syn_1))
    error = layer_2 - y_truth
    layer_2_delta = error * sigmoid_output_to_derivative(layer_2)
    layer_1_error = layer_2_delta.dot(syn_1.T)
    layer_1_delta = layer_1_error * sigmoid_output_to_derivative(layer_1)
    syn_1 -= layer_1.T.dot(layer_2_delta)
    syn_0 -= X_XOR.T.dot(layer_1_delta)

print("Output After Training: n", layer_2)
```

反向传播，矩阵乘法和梯度下降放在一起会让人很难理解。这个过程的可视化通常是对其背后原理的简化。专注于理解其背后的逻辑，但不要过多地考虑直觉上的理解。

另外，读者们也可以看看 Andrew Karpathy 关于反向传播的课程，在这个可视化网站交互学习，以及阅读 Michael Nielsen 关于反向传播的章节。

深度神经网络

深度神经网络就是在输入层和输出层之间具有多个中间层的神经网络。这个概念最早是由 Rina Dechter (Dechter, 1986) 引入的，但在 2012 年，也就是在 IBM 的人工智能程序 Watson 赢得美国电视智力竞赛节目 Jeopardy 和 Google 推出猫咪识别器之后才受到广泛关注。

深度神经网络与之前神经网络的核心结构相同，但是应用于一些不同的问题。在正则化方面也有很大改进。最初 ,这只是一组用来简化冗杂的地球数据的数学函数(Tikhonov，A.N.，1963）。而现在被用于神经网络中，以加强其泛化能力。

这种技术创新很大程度上依赖于计算机的运算能力。而运算能力的提升大大缩短了研究者的创新周期 —— 如今的 GPU 技术只需半秒钟就可以完成一个八十年代中期的超级计算机一年的运算量。

计算成本的降低和各种深度学习库的发展将深度学习带入了大众视野。我们来看一个常见的深度学习堆栈示例，从底层开始：

- GPU > Nvidia Tesla K80。该硬件常用于图形处理。它们深度学习的速度平均要比 CPU 快 50-200 倍。
- CUDA > GPU 的底层编程语言
- CuDNN > Nvidia 的库，用来优化 CUDA
- Tensorflow > 由 Google 开发，基于 CuDNN 的深度学习框架
- TFlearn > Tensorflow 的前端框架

下面我们来看看 MNIST 数字分类图像，它被称作深度学习的 “Hello World” 。

我们用 TFlearn 来实现：

```Python
from __future__ import division, print_function, absolute_import
import tflearn
from tflearn.layers.core import dropout, fully_connected
from tensorflow.examples.tutorials.mnist import input_data
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.normalization import local_response_normalization
from tflearn.layers.estimator import regression

# Data loading and preprocessing
mnist = input_data.read_data_sets("/data/", one_hot=True)
X, Y, testX, testY = mnist.train.images, mnist.train.labels,
mnist.test.images, mnist.test.labels
X = X.reshape([-1, 28, 28, 1])
testX = testX.reshape([-1, 28, 28, 1])

# Building convolutional network
network = tflearn.input_data(shape=[None, 28, 28, 1], name='input')
network = conv_2d(network, 32, 3, activation='relu', regularizer="L2")
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = conv_2d(network, 64, 3, activation='relu', regularizer="L2")
network = max_pool_2d(network, 2)
network = local_response_normalization(network)
network = fully_connected(network, 128, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 256, activation='tanh')
network = dropout(network, 0.8)
network = fully_connected(network, 10, activation='softmax')
network = regression(network, optimizer='adam', learning_rate=0.01,
                     loss='categorical_crossentropy', name='target')

# Training
model = tflearn.DNN(network, tensorboard_verbose=0)
model.fit({'input': X}, {'target': Y}, n_epoch=20,
           validation_set=({'input': testX}, {'target': testY}),
           snapshot_step=100, show_metric=True,
run_id='convnet_mnist')
```

关于 MNIST 问题，有很多不错的文章：

https://www.tensorflow.org/get_started/mnist/beginners

https://www.youtube.com/watch?v=NMd7WjZiCzc

https://www.oreilly.com/learning/not-another-mnist-tutorial-with-tensorflow

如果读者想更深入地了解 TFlearn，可以浏览我之前的文章。

小结

我们从 TFlearn 的示例中可以看出，深度学习的基础逻辑仍与 Rosenblatt 的感知机类似。 如今的神经网络大多使用 Relu 激活函数，而不是二元 Heaviside 阶跃函数。在卷积神经网络的最后一层，损失函数使用的是 categorical_crossentropy，即分类交叉熵。这是对 Legendre 最小二乘法的改进，可用于多分类逻辑回归问题。这里的优化算法 adam 来源于 Debye 的梯度下降。Tikhonov 的正则化概念以 dropout 层和正则化函数 L1/L2 的形式得到广泛应用。

如果您想要对神经网络的概念以及如何实现有一个更直观的理解，请阅读我在 FloydHub 博客上发表的文章：我的第一个深度学习周末。

感谢 Ignacio Tonoli，Brian Young，Paal Ringstad，Tomas Moska 和 Charlie Harrington 帮我审阅本文的草稿。代码来源储存在 Jupyter notebook 中。

关于 Emil Wallner

这是 Emil 博客中深度学习系列的第二部分。Emil 花了十年时间探索人工学习。他曾在牛津大学商学院工作，投资教育创业公司，并开创了了技术教学业务。去年，他加入 Ecole 42，并将自己在人工学习领域的知识应用于机器学习。