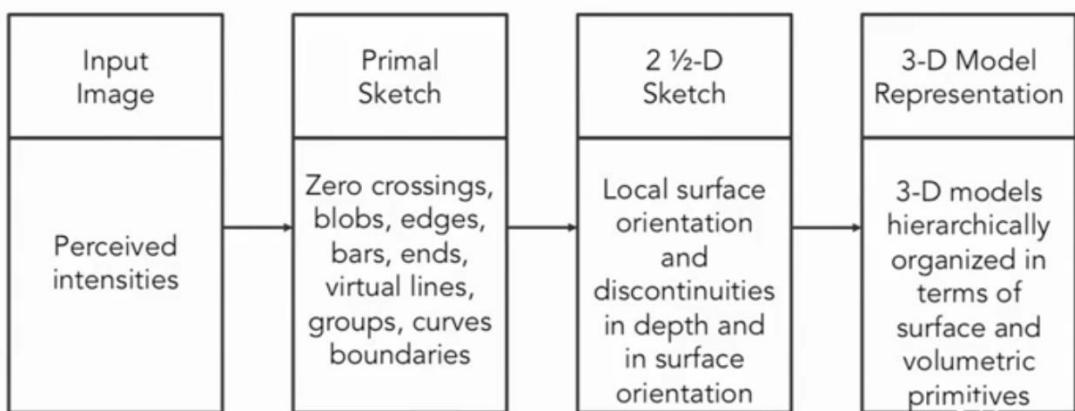
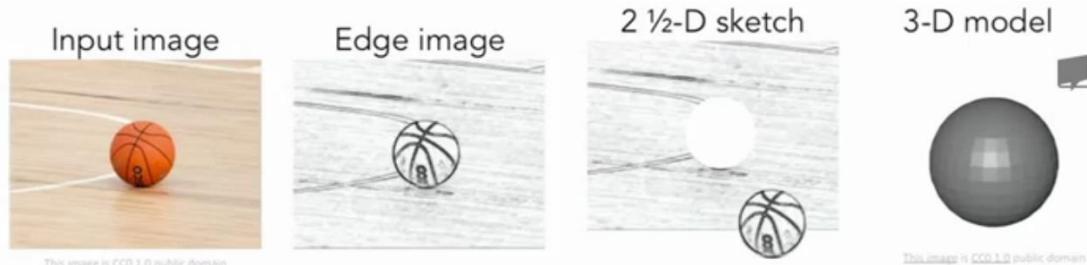
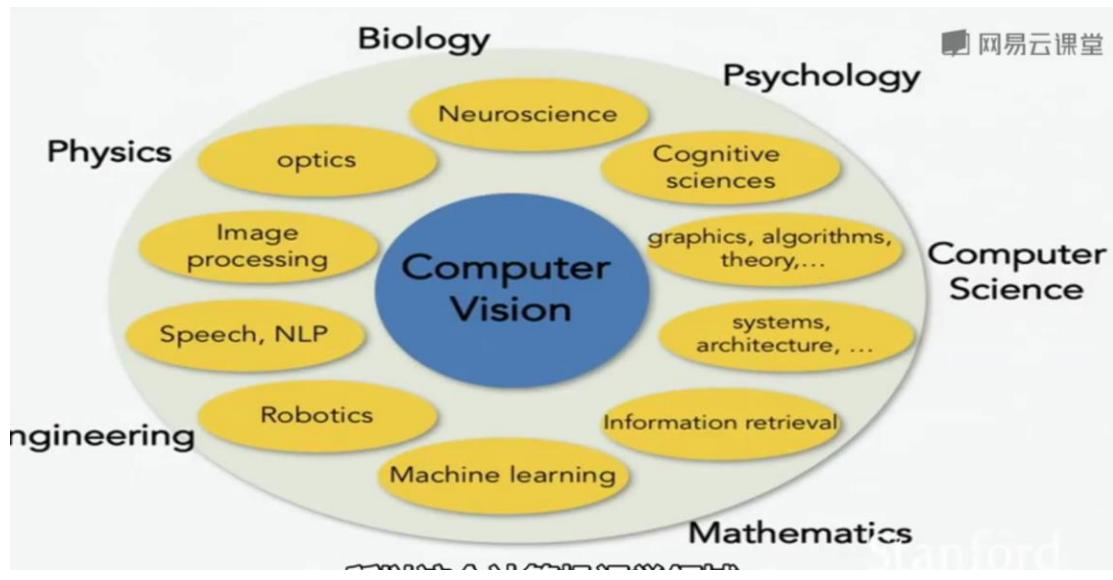
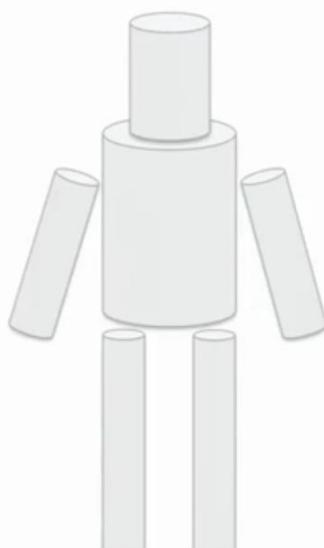


CS231n



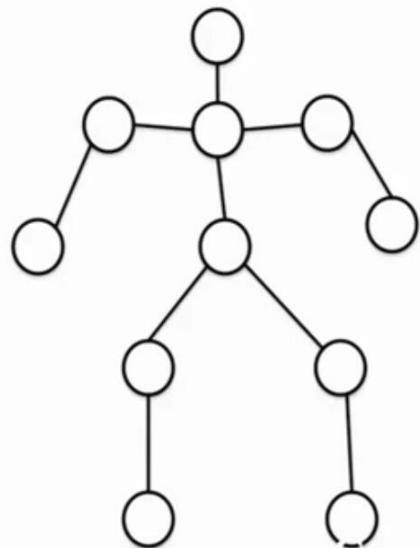
- Generalized Cylinder

Brooks & Binford, 1979



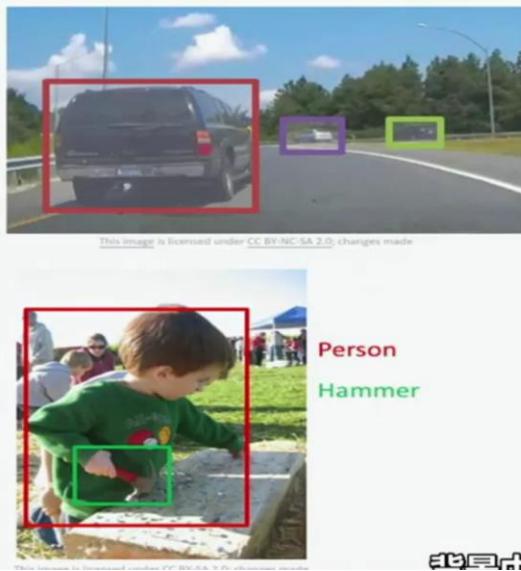
- Pictorial Structure

Fischler and Elschlager, 1973

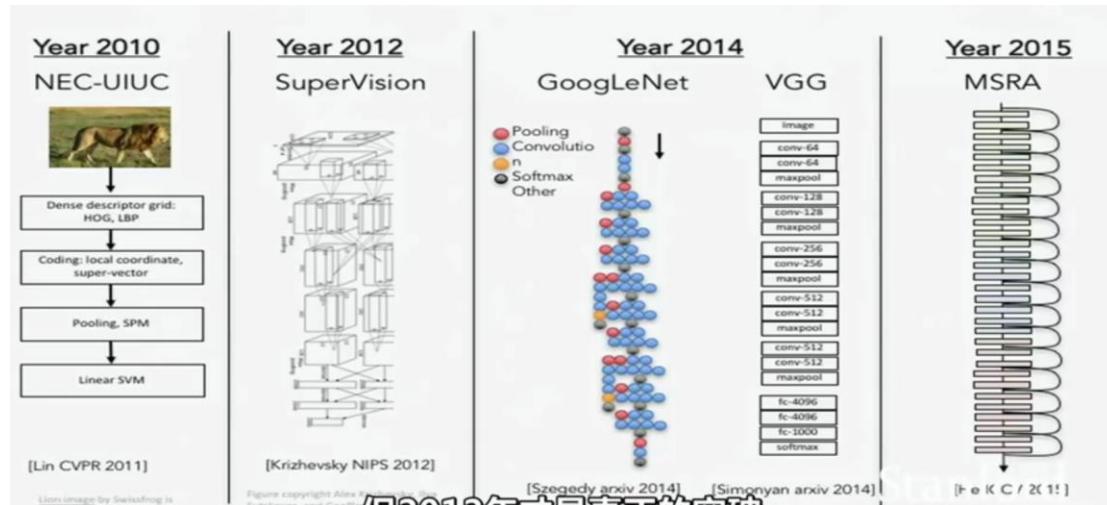


概述

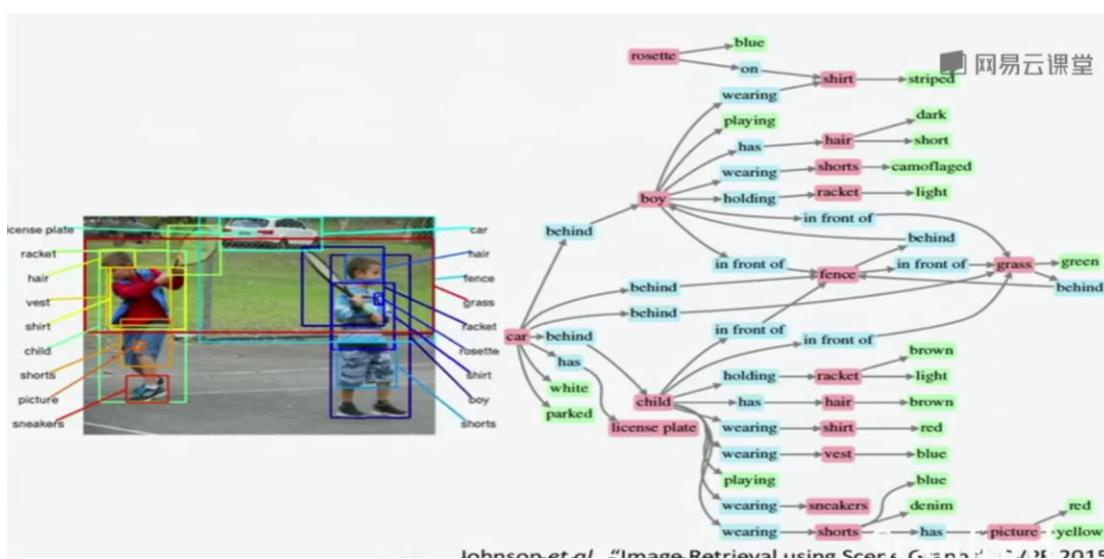
图像分类



- Object detection
 - Action classification
 - Image captioning
 - ...



在 2012 年提出了卷积神经网络 AlexNet，从此图像识别基本就是卷积神经网络的改进。



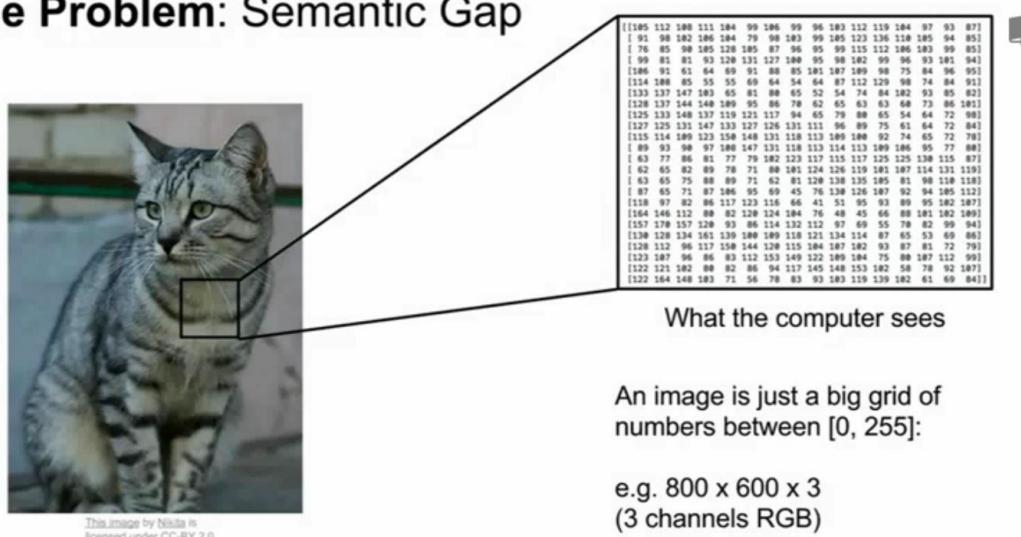
图像分类—数据驱动方法

Python+numpy

语义鸿沟

我们赋予图片中的猫一个语义概念，但是这个语言概念跟计算机实际能看到的像素值之间有很大的差距

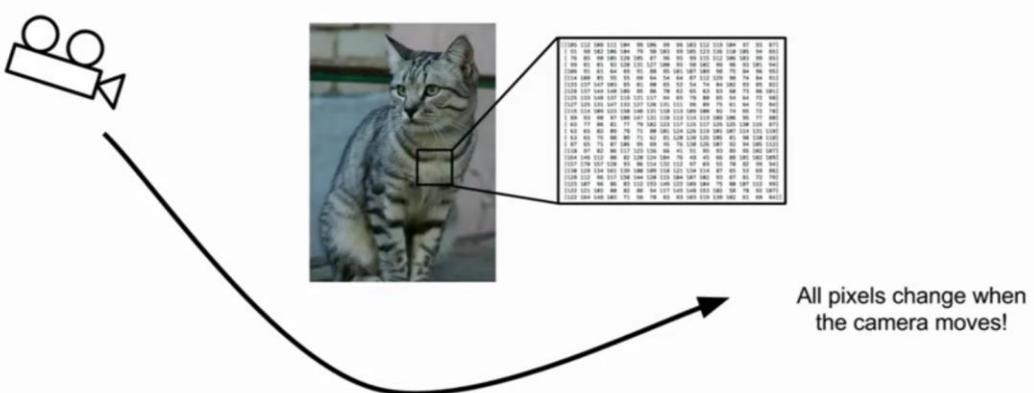
The Problem: Semantic Gap



我们可以对图片做出很微小的改动，这就使得像素值发生了很大的变化。比如我们将相机移动，那就使得像素值发生了相当大的变化，当然我们看到的仍然是同一只猫，因此我们需要对这些变化进行鲁棒（抗干扰）

Challenges: Viewpoint variation

网易云课堂



不仅是这种变化，在不同的光照环境下，无论猫出现在哪种光照下，我们都应该辨识出他，这也要求我们的算法对此有鲁棒性。

Challenges: Illumination

网易云课堂



This image is CC0 1.0 public domain



This image is CC0 1.0 public domain



This image is CC0 1.0 public domain



This image is CC0 1.0 public domain

有时候目标对象会发生形变，我们也要对此有鲁棒性

Challenges: Deformation

网易云课堂



This image by Umberto Salvagnin
is licensed under CC-BY 2.0



This image by Umberto Salvagnin
is licensed under CC-BY 2.0



This image by sara.bear is
licensed under CC-BY 2.0



This image by Tom.Thai is
licensed under CC-BY 2.0

遮挡也会影响计算机的像素值，但我们仍然能够辨识出来这是一只猫，这也是我们算法需要鲁棒地应对的一种情况

Challenges: Occlusion

网易云课堂



This image is CC0 1.0 public domain



This image is CC0 1.0 public domain



This image by jessica is licensed
under CC-BY 2.0

猫也有可能和背景混合在一起，这也是我们需要处理的问题

Challenges: Background Clutter

网易云课



类内差异，虽然图片中都是猫，但是每个对象有不同的形状、大小、颜色和年龄。算法需要处理这些不同。

Challenges: Intraclass variation

网易



An image classifier

网易云课

```
def classify_image(image):
    # Some magic here?
    return class_label
```

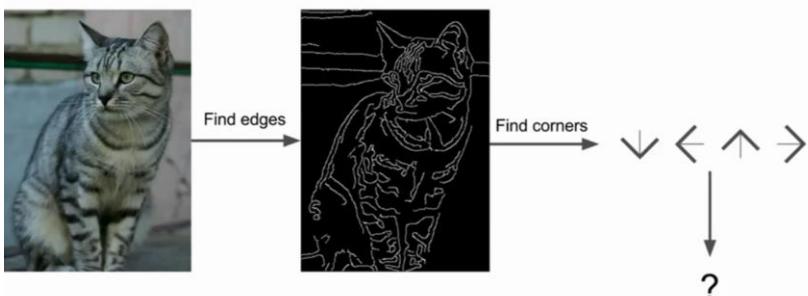
Unlike e.g. sorting a list of numbers,

no obvious way to hard-code the algorithm for
recognizing a cat, or other classes.

我们将图片的边缘提取处理，将各种形状分类好，通过某些已有的我们认为的猫的特征来识别出猫。这种方法的准确率堪忧，并且我们要识别另外一个对象时，我们就要重新进行这个流程，总的来说，这种方法并不是一种可以推演的方法。上面是这种方式的 API，我们输入图片，得到这个类的标签。

Attempts have been made

网易云课堂



我们想要找到一种可以识别所有物体的算法，因此我们想到了数据驱动的方法

Data-Driven Approach

网易云课堂

1. Collect a dataset of images and labels
2. Use Machine Learning to train a classifier
3. Evaluate the classifier on new images

Example training set

```
def train(images, labels):
    # Machine learning!
    return model

def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```



我们不再具体的描述一只猫，而是从网上抓取大量的猫的图片数据集或者任意我们想要识别的对象的图片。我们得到了这些图片数据集，然后我们训练机器来分类这些图片，机器会收集所有数据，并且用某种方式来总结，然后生成一个模型，总结出识别出这些不同类的对象的核心知识要素，然后我们用这哥模型来识别新的图片中同类对象。这样的话，我们的接口就会变成—我们有两个函数，一个是训练函数，这个函数用于接受图片和标签，然后输出模型。另一个函数是预测函数，接收一个模型，对图片种类进行预测。就是利用这种方式，在过去的十几二十年间，图像识别领域的进步非常大。

CS231 是关于神经网络的课程，但是这种数据驱动类的算法是比深度学习更为广义的一种理念，通过这种理念，对于一个简单的分类器，我们不需要做任何事情就可以得到最近邻分类器，只需记录数据，在预测函数中，我们输入新的图片，然后得到标签。这种思想是很简单的，但是这里用到的很多属性，但是数据驱动的。

```
def train(images, labels):
    # Machine learning!
    return model
```

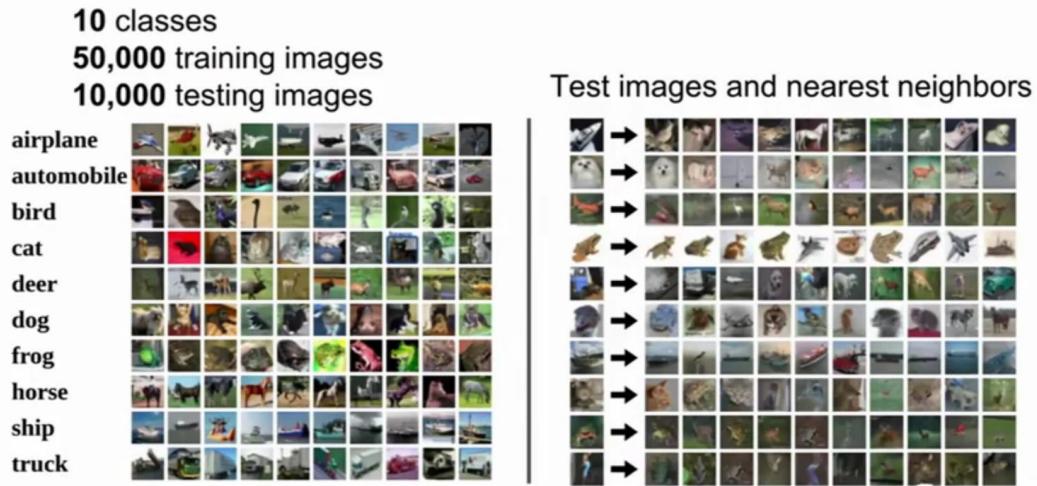
Memorize all
data and labels

```
def predict(model, test_images):
    # Use model to predict labels
    return test_labels
```

Predict the label
of the most similar
training image

来看最近邻分类器，虽然分类的对象有可能不是一类，但是确实图片中都是很相似的。我们将最近邻分类器应用于数据集，那我们就能在训练集中找到最接近的示例（样本），因为他们来自于数据集，因此我们知道这些最接近示例的标签，这是我们可以简单的指出，这幅测试图片也是一只狗。尽管分类的结果并不是很理想，但它仍然是一个很好的示例。

Example Dataset: CIFAR10



© Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009

如果我们给出两张图片，我们应该如何比较它们？因为我们需要将测试图片和所有训练图片进行比较，我们有很多不同的选择来确定比较的函数。

我们在测试 1 中使用了 L1 距离，或者说是曼哈顿距离，我们将图片的单个像素进行比较，我们将测试图片的左上角像素和训练图片的左上角像素进行差值计算后取绝对值，然后对每个像素都做这样的操作后将其值求和。这样处理显得笨拙，但有些时候却也有它的合理性。

$$\text{L1 distance: } d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

test image				training image				pixel-wise absolute value differences					
56	32	10	18	-	10	20	24	17	=	46	12	14	1
90	23	128	133		8	10	89	100		82	13	39	33
24	26	178	200		12	16	178	170		12	10	0	30
2	0	255	220		4	32	233	112		2	32	22	108

add → 456

下面是最近邻分类器的 PYTHON

```

import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred

```

模型训练非常简单，因为使用了 numpy, 我们只要需要存储训练数据

```

def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

```

Memorize training data

对于每个训练图片：通过 L1 距离找到最临近训练图片，然后预测最邻近图片的标签

```

# loop over all test rows
for i in xrange(num_test):
    # find the nearest training image to the i'th test image
    # using the L1 distance (sum of absolute value differences)
    distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
    min_index = np.argmin(distances) # get the index with smallest distance
    Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

```

For each test image:
Find closest train image
Predict label of nearest image

对于 N 个实例，训练和预测速度如何？

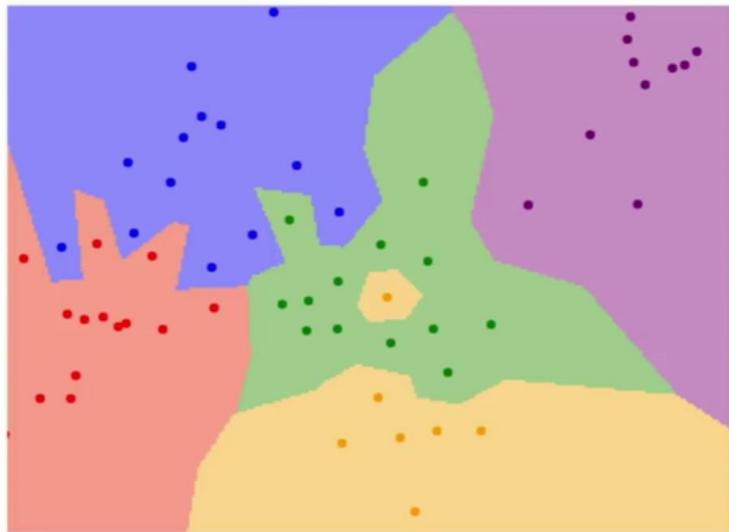
Q: With N examples,
how fast are training
and prediction?

A: Train O(1),
predict O(N)

This is bad: we want
classifiers that are **fast**
at prediction; **slow** for
training is ok

可以看到，训练时间是较快的，而预测时间要慢得多，这与我们的期望不符，我们都期望能够训练时间长，而预测时间短。我们在服务器上训练模型，而在低性能的设备上预测对象。那么我们得到一个结论，最邻近算法已经过时了，而神经网络等算法则是我们所期望的

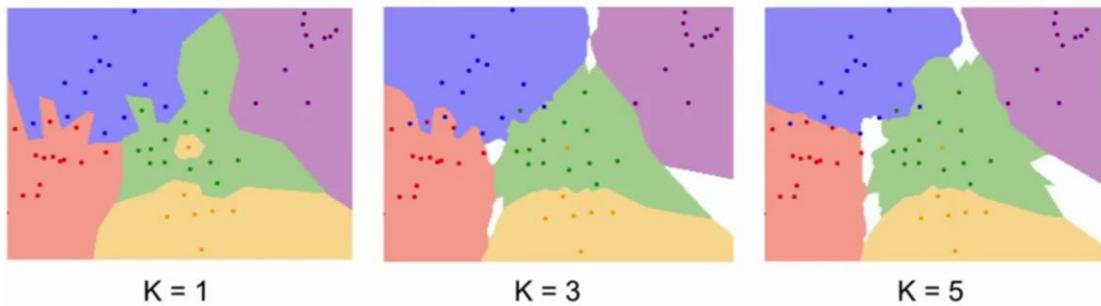
What does this look like?



最邻近分类器的决策区域，我们可以看到，图片中心的黄色噪点由于只测量最近 L2 距离使得绿色区域中有一块黄色区域，并且我们可以看到，绿色区域深入了图片上方，这有可能是噪点引起的。

K-Nearest Neighbors

Instead of copying label from nearest neighbor,
take **majority vote** from K closest points



为解决最邻近分类器的问题，产生了 k-临近算法，它不只是寻找最近的点，根据我们的距离度量，找到最近的 k 个点，然后再这些相邻点中进行投票，然后票数多的邻近点预测出结果。我们可以看到，在 $K=3$ 时，即使有黄色的噪点，他也没有影响到绿色的划分，并且绿色区域不再出现明显的突出。因此我们在使用最邻近算法时，我们通常会给较大的 k 值，使得决策区域更为光滑，结果更好。

图中的白色区域没有获得 k 投票，我们也可以将其视为一个类别。

通过下面的结果，我们可以看到 KNN 的效果不尽如人意，如果我们将 K 的值放大，那么搜索的范围可能是 3 张 4 张甚至所有数据。当我们使用这种方法来检索相邻数据时，这样就会对噪声有更大的鲁棒性。



也有另外一种做法，在我们使用 K-近邻算法时，对于不同的图片，确定我们应该如何比较相对近邻数据距离值，在上面我们使用的方法是 L1 距离，是像素差距的绝对值的求和。而另一种常见的选择是 L2 距离，也就是欧式距离，即取平方和的平方根并将其作为距离。选择不同的距离，会在你预测的空间里，对底层的几何或者拓扑结构做出不同的假设。

如下图所示，根据 L1 距离，这个围绕着原点的方形形成的圆，在这个方形上，在 L1 上是跟原点等距的；而在 L2 距离或是欧式距离，类似的形成的是一个圆，更像是我们所期望的。这两个距离显示了一些东西，L1 距离依赖于坐标轴，如果你转动坐标轴，将会改变点之间的 L1 距离，而改变坐标轴则对 L2 距离毫无影响。所以，如果你输入的向量有一些重要的意义，那么 L1 有可能更适合。但如果他只是某个空间中的一个通用向量，而你却不知道其中的不同的元素，不知道他们的意义如何，那么 L2 可能会更自然一点。另一点更重要的，采用不同的距离度量，我们可以将 K-临近法泛化到许多不同的数据类型上，不仅是数据和图像。

K-Nearest Neighbors: Distance Metric

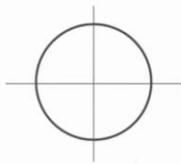
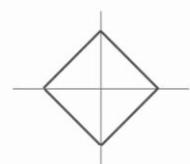
回顶部

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



当我们将两种不同的距离度量应用到 K-临近分类器中时，我们可以看到区域的边界发生的变化很大，L1 距离的分割区域更趋向于跟随坐标轴，这又是因为 L1 取决于我们选择对坐标系统的选定，而 L2 对距离的排序并不会受到坐标轴的影响，他只是把边界放置在存在最自然的地方。

K-Nearest Neighbors: Distance Metric

网易云课

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



K = 1

L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



K = 1

我们讨论了如何选择 K 和如何选择距离度量这样的问题，这些参数我们称之为超参数。因为它们不一定都能从训练数据中学到，我们需要提前为算法选择这样的参数，那么我们应该如何在实践中确定这样的参数？

Hyperparameters

网易云课

What is the best value of k to use?

What is the best distance to use?

These are **hyperparameters**: choices about the algorithm that we set rather than learn

Very problem-dependent.

Must try them all out and see what works best.

也许我们的第一个想法是选择让我们的数据集准确率最高的超参数，这是一种错误的想法，在机器学习中，我们追求的是在训练集以外的未知数据上表现更好，

Setting Hyperparameters

网易云

Idea #1: Choose hyperparameters that work best on the data

BAD: K = 1 always works perfectly on training data

Your Dataset

另外一种想法可能是：我们在训练数据集上使用不同的超参数，然后将得到的分类器应用到测试数据集上，然后我们选择在测试数据集中效果最好的那组超参数。这种想法也是错误的，测试数据集是给我们一种预估方法，即在没遇到的数据上算法表现将会如何，如果采用这种用不同的超参数训练不同算法的策略，然后选择在测试时表现最好的超参，那么我们就有可能只是选择了一组超参只是在这组测试集上表现良好，但是在这组测试数据集上的表现无法代表在全新的未见过的数据上的表现

Idea #2: Split data into **train** and **test**, choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data

train

test

更合理的做法是将数据集分为三部分，其中大部分数据作为训练集，然后建立一个验证集和一个测试集。我们要做的就是在训练集上用不同超参来训练算法，在验证集上进行验证，然后选择一组在验证集上表现最好的超参。这时，所有的调试就完成了，再将这组在验证机上表现最佳的分类器拿出来，在测试集上跑一跑，这才是你要写在论文和报告上的数据。这个数据才是你在未知的数据集上表现成绩。很重要的一点是，我们必须分隔验证集和测试集，所以，当我们做研究报告时，我们通常都是在最后一刻才接触测试集，

Idea #3: Split data into **train**, **val**, and **test**; choose hyperparameters on **val** and evaluate on **test**

Better!

train

validation

test

设定超参的一种方式是交叉验证，在小型的数据集中更常用一些，在深度学习中并不常用。它的理念：我们将所有的数据集保留一部分数据作为最后使用的测试集，对于剩余的数据集，我们并非将其分为一个训练集和一个验证集，而是将训练集分成很多份，在这种情况下，我们轮流将每一份都当作验证集。如下所示，我们先将前四份作为训练集，第五份作为验证集。然后再在 1、2、3、5 上进行训练，在第 4 份上进行验证，然后对不同份进行循环，这么做似乎是很合理的，我们可以更确定哪组超参更为稳定。但事实上，在深度学习中，训练本身十分消耗计算能力，因此这些方法实际上并不是那么实用。

Idea #4: Cross-Validation: Split data into **folds**, try each fold as validation and average the results

fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test
fold 1	fold 2	fold 3	fold 4	fold 5	test

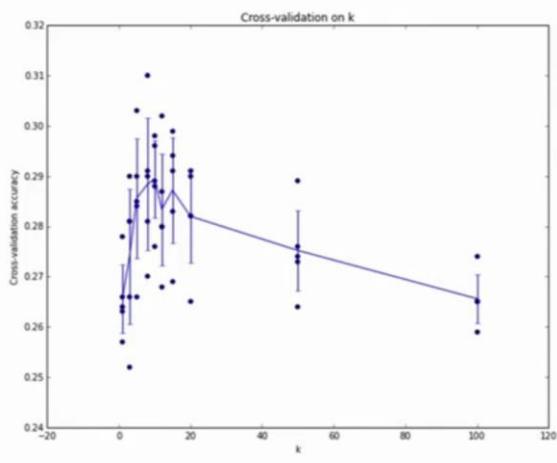
Useful for small datasets, but not used too frequently in deep learning

训练集和验证集的区别？

以 KNN 为例，训练集是一堆贴上标签的图片，我们记下标签来给图片分类，我们会将图片和训练集的每一个元素进行比较，然后将与训练点最接近点的标签作为结果，我们的算法会记住训练集中的所有样本，然后我们会把验证集中的每个元素与训练集中的每个元素比较，将它作为依据来判定分类器的准确率，在验证集上的表现如何。这就是训练集和验证集的区别。你的算法可以看到训练集中的各个标签，但是在验证集中，你的算法不能直接看到他们的标签。我们只是用验证集中的标签来检查我们算法的表现。

我们使用了 5 折交叉验证，也就是对于每个 K 值，我们都对算法进行了 5 次不同的测试

Setting Hyperparameters



Example of
5-fold cross-validation
for the value of k .

Each point: single
outcome.

The line goes
through the mean, bars
indicated standard
deviation

(Seems that $k \approx 7$ works best
for this data)

KNN 在图像中应用很少，原因有几个，首先是他测试时运算时间很长，其次是 L2 或者 L1 距离这样的衡量标准，用在比较图像上是在不太适合，这种向量化的距离函数，不太适合表示图像之间视觉的相似度。

k-Nearest Neighbor on images never used.

网易云课堂

- Very slow at test time
- Distance metrics on pixels are not informative



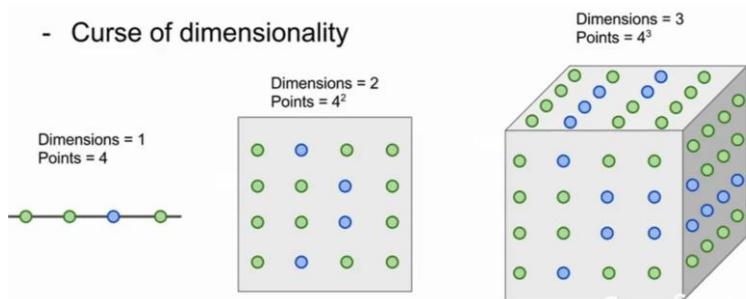
all images in
public domain

(all 3 images have same L2 distance to the one on the left)

维度灾难，KNN 是将数据分为几部分，这样就要求我们的数据密集的分布在空间中，否则最近邻点的实际距离可能很远，也就是说和待测样本的相似性没有那么高。为了保证密集分布，我们需要指数倍的训练数据，数据量巨大并且难以获取这样的高维空间内的像素。

我们可以看到，有可能在一维空间内 4 个点就能形成密集的空间分布，二维则需要 4^2 个，三维需要 4^3 个，如果空间越高维，则需要的数据量就成指数倍的增长，这不是我们所乐见的。

- Curse of dimensionality



K-Nearest Neighbors: Summary

网易云课堂

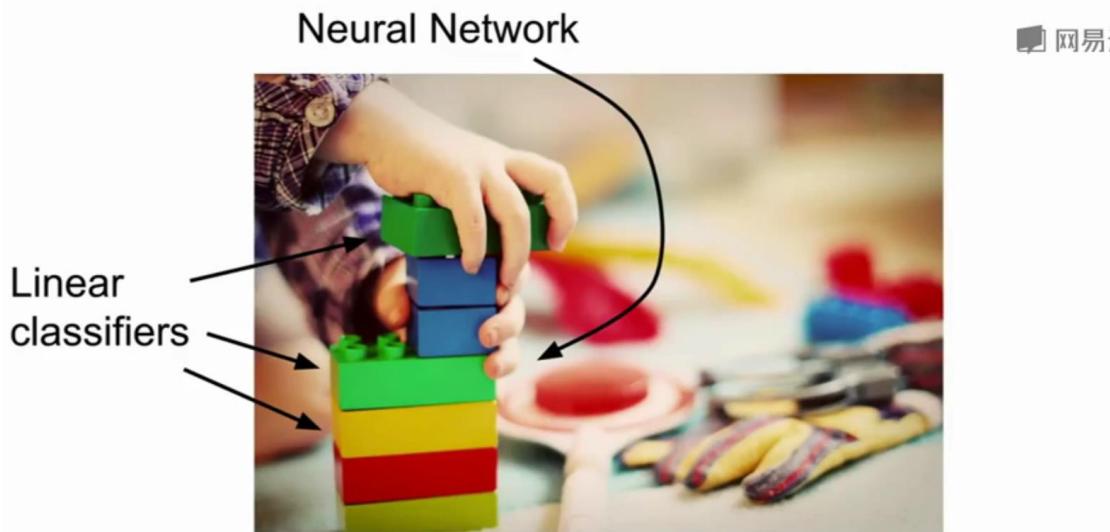
In **Image classification** we start with a **training set** of images and labels, and must predict labels on the **test set**

The **K-Nearest Neighbors** classifier predicts labels based on nearest training examples

Distance metric and K are **hyperparameters**

Choose hyperparameters using the **validation set**; only run on the test set once at the very end!

线性分类



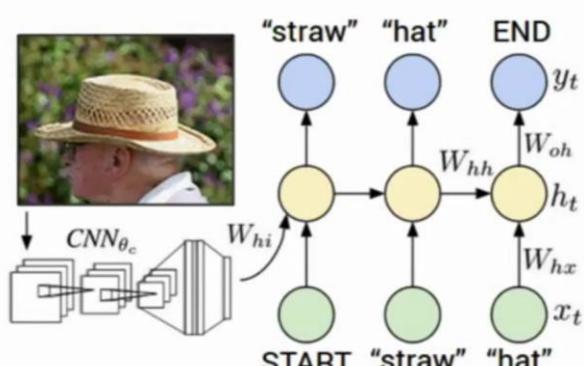
Two young girls are playing with lego toy. Boy is doing wakeboard. 雷锋字幕组招募中 申请加微信 julylihuaijiang

网易云课堂



Man in black shirt is playing guitar.

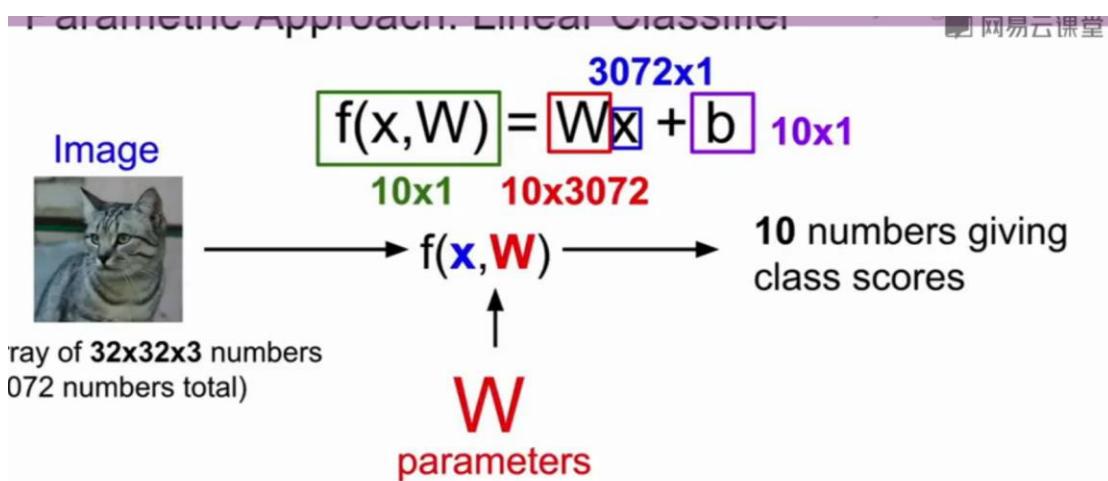
Construction worker in orange safety vest is working on road.



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

线性分类器是参数模型中最简单的例子。现在，我们的参数模型实际上有两个不同的部分，以该图为例，可能是，一只猫在左边还有一组参数，我们通常把输入数据写为 X，将参数设置或称为权重通常叫做 w,有时也叫 theta。现在我们这个函数包含了输入数据 x 和参数 W,

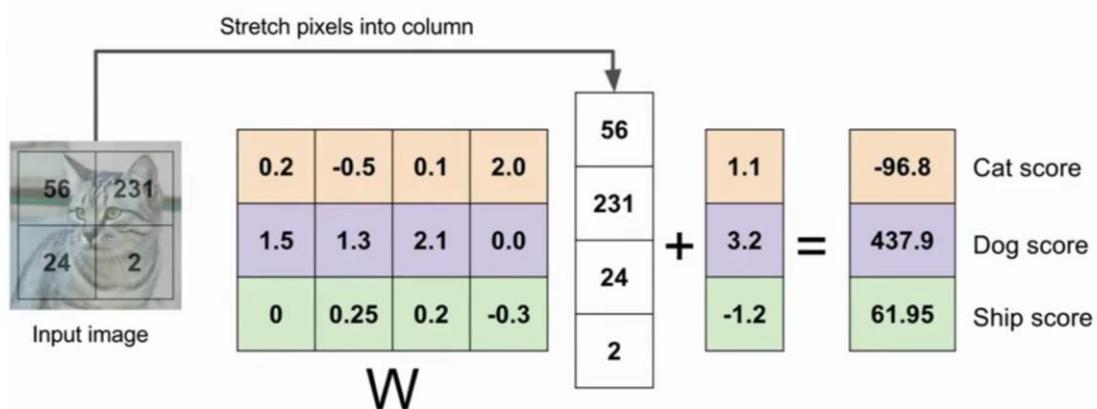
这是就会出现十个数字来描述在 CIFAR-10 中对应的 10 个类别所对应的分数。根据上面的表述，比如说猫的分数更大，则表明输入 X 是猫的可能性更大。使用这种方法，我们可以丢弃训练集，而只保留 W ，从而使模型效率大大提高，甚至可以运行在手机这样的设备上。因此，在深度学习中，整个描述都是关于函数 F 的正确结构，我们可以使用各种不同的函数结构来组织 F ，但相乘是个很容易的想法，这就是一个线性分类器，所以 $f(x, W) = Wx$ 可能是你能想到的最简单的方程



现在我们有一张 2*2 像素的图片，我们将这张图片拉伸成一个有四个元素的列向量，我们的权重矩阵为 4（像素数）*3（类别数—cat dog ship）的，然后是一个 3*1 的偏差向量，他给我们提供了每个类别的数据独立的偏差项。

我们可以理解线性分类，几乎是一种模板匹配方法，这个矩阵中的每一行对应于图像的某个模板。现在，根据输入矩阵行和列之间的乘积或点积从而得到图像的像素点，计算这个点积让我们发现这个模板和图像像素之间有一个相似之处，然后再一次，偏差给了你这个数据独立缩放比例以及每个类的偏移量，我们根据这个模板匹配的观点考虑线性分类器。实际上，我们可以取那个权重矩阵的行向量并且将它们还原回图像。

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

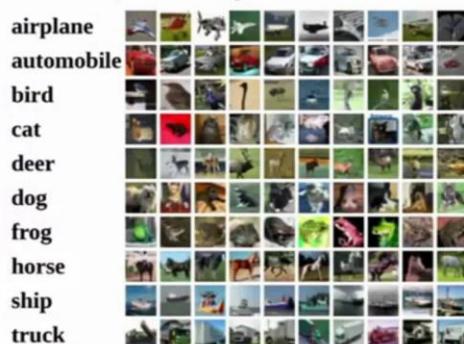


实际上是将这些模板可视化成图像，这就给我们展示了一个线性分类器对于我们的数据实际上可能是怎样做的。如下图所示，下面是我们已经训练好的权重矩阵的行向量所代表的十个类别的模板的可视化结果。举一例，airplane 的代表模板可视化是蓝色，有一些斑点状的像素，这样的话，我们就可以认为，这个行向量再找具有这些特征的图片。这就出现了一个问题，线性分类器每个类别只能学习一个模板，如果这个类别出现了某种类型的变体，他就

会尝试求取所有不同变体，所有那些不同变现的平均值，并且只使用一个单独的模板来识别其中的每一个类别。因为线性分类器只允许学习一个类别的一个模板，而如果是神经网络等方法，我们就能够得到更好的准确率，因为这些方法在于没有每个类别学习一个单独的模板的限制。

Interpreting a Linear Classifier

网易云课堂



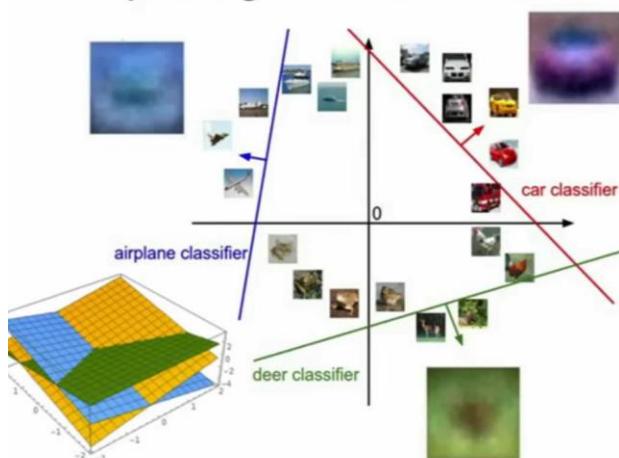
$$f(x, W) = Wx + b$$

Example trained weights
of a linear classifier
trained on CIFAR-10:

线性分类器的另一个概念是回归图像，作为点和高维空间的概念，你可以想象成我们每一张图像都是类似于高维空间中一个点的东西，现在线性分类器尝试在线性决策边界上尝试画一个线性分类面来划分一个类别和剩余其他类别，如下坐标图所示。

Interpreting a Linear Classifier

网易云课堂



$$f(x, W) = Wx + b$$



Array of 32x32x3 numbers
(3072 numbers total)

但是如果从高维坐标的角度来考虑线性分类器，你就能再次看到线性分类器中可能出现的问题。举一例，下面左图，我们将像素索引大于 0 且为奇数的像素设为蓝色，将其他的设为红色，总共有两类，如果你去画这些不同的决策，出现不同的决策取悦，你能看到我们奇数像素点的蓝色类别，在平面上有两个象限，甚至是两个相反的象限，因此我们没有办法能够绘制一条单独的直线来划分红色和蓝色，这是线性分类器的困境。也许归根结底这不是人工数据，实际上我们是在计算图像中动物或者人类的奇偶数，而非像素点。所以这种奇偶数划分的问题是线性分类器通过传统方法难以解决的。另外一个难以解决的是多分类问题，我们可以看到，在下图右，蓝色存在于三个像素，然后其他所有部分是另外一个类别。有可能出现我们前面出现的问题，训练集中一匹马的头向左看，而另一图片中的马向右看，在得到的权重 W 的可视化结果中，就可能得到一个模糊的有两个头的马匹，而这样的情况，我们没

有办法通过一条线将其分类开。所以，任何时候当你有多模态数据，比如一个类别出现在不同的领域空间中，这是另一个线性分类器可能有困境的地方。虽然存在着诸多问题，但线性分类器是一个很简单的算法，易于使用和理解

Hard cases for a linear classifier



So far: Defined a (linear) score function $f(x, W) = Wx + b$

Example class scores for 3 images for some W :



How can we tell whether this W is good or bad?

airplane	-3.45	-0.51	3.42
automobile	-8.87	6.04	4.64
bird	0.09	5.31	2.65
cat	2.9	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	-4.34
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.11

权重矩阵 W 中的每一行，都代表了一类对象的模板，我们可以这么认为：我们的输入 X 是我们需要测试的对象，而他会与每个类的模板做乘法----我们可以将其看作一个比较的过程，我们拿这个对象分别去和每个类的模板做比较，然后我们得到一个比较的结果，这个结果和相似程度正相关，对象和某个类的比较结果越大，那他就越类似于这个类，对于值最大的类我们就可以这么认为它是属于这个类的。我们将 W 中的一行转置为列向量，则它就是一个做了拉伸像素处理的对象，我们将其还原为原本的像素结构，那就可以得到可视化的类的模板。另一个解释：学习像素在高维空间的一个线性决策边界，其中高维空间就对应了图片能取到的像素密度值。

损失函数和优化介绍

Recall from last time: Linear Classifier

网易云课堂



airplane	-3.45	-0.51	3.42
automobile	-8.87	6.04	4.64
bird	0.09	5.31	2.65
cat	2.9	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	-4.34
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

TODO:

- Define a **loss function** that quantifies our unhappiness with the scores across the training data.
- Come up with a way of efficiently finding the parameters that minimize the loss function. (**optimization**)

损失函数：可以用一个函数把 W 当作输入，然后观察得分，定量的估计 W 的好坏，这个函数就称为损失函数。

优化：我们要通过损失函数来在 W 可选择的范围内选择不是最差的 W ，这个过程将是一个优化过程。

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

A loss function tells how good our current classifier is

Given a dataset of examples
 $\{(x_i, y_i)\}_{i=1}^N$

Where x_i is image and y_i is (integer) label

Loss over the dataset is a sum of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

损失函数：有一些训练数据集 x 和 y ，通常我们说有 N 个样本，其中 x 是算法的输入（在图像分类问题中， x 其实是图片每个像素点所构成的数据集），而 y 则是我们希望算法预测出来的东西（我们通常称为标签和目标）。具体一点，我们尝试把 CIFAR-10 中的每个图片都分到十个类别中的某一类中，所以 y 是一个在 1-10 之间（0-9 也一样）的整数，总之 y 表示了对于每个图片 x 哪个类才是正确的结果。我们给出损失函数 L 的定义：通过函数 f 给出预测的分数和真实的目标（或者说标签 y ），可以定量地描述训练样本预测的好不好。最终的损失函数 L 是整个数据集中 N 个样本的损失函数的总和的平均。这样的思想可以应用到非图像问题上，用于衡量参数 W 是否让人满意，然后再所有能取到的 W 中，找到在训练集上损失函数极小化的 W 。

多类别 SVM 损失函数

是在处理多分类时候的一种推广

Multiclass SVM loss

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the
scores vector: $s = f(x_i, W)$

the SVM loss has the form:

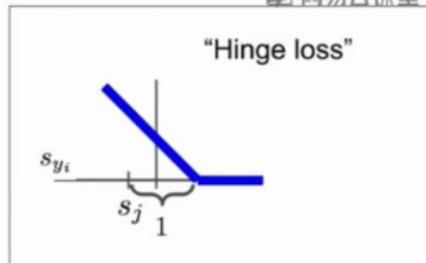
$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

除了正确的分类 Y_i 以外，对所有的分类 Y 都做加和，也就是说

我们在所有错误的分类上做和，比较正确分类的分数和错误分类的分数。如果正确分类的分数比错误分类的分数高，比错误分类的分数高出某个安全的边距，我们将这个边际设为 1，那么损失就为 0，接下来把图片对每个错误分类的损失加起来，就可以得到数据集中这个样本的最终损失，还是同样的对整个训练集取平均。这有点类似于 if then 语句。

Multiclass SVM loss



$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

来看这个合页函数图像，竖轴为损失 L_i ，横轴为所得分

数，当正确类的分数小于(错误类的分数+1)，我们就取其插值作为 L_i ，而如果正确类的分数大于 (错误类的分数+1)，我们取 L_i 为 0，如图像所示， L_i 现时线性的减少，到达 0 之后不再减少。

对符号做一个解释：S 是通过分类器预测出来的类的分数， Y_i 代表这个样本的正确的分类标签， S_{Y_i} 代表了训练集的第 i 个样本的真实分类的分数，而 S_i 则是代表了除了正确分类外的别的类的得分。总的来说，我们某个样本，在他正确的分类上获得的分数大于他在其他的错误的分类上获得的分数+1，那对于这个样本，我们训练得到的 W 的损失就为 0，否则就是 $(S_j - S_{Y_i} + 1)$ (所有错误的分类的分数-正确类分数+1) 的和。

With some W the scores $f(x, W) = Wx$ are:



Given an example (x_i, y_i)

where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the
scores vector: $s = f(x_i, W)$

cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1
Losses:	2.9		

如果我们对第一个训练样本 $2.9 + 0$
 $\sqrt{40}$ and now if we look at, if we think

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \max(0, 5.1 - 3.2 + 1)$$

$$+ \max(0, -1.7 - 3.2 + 1)$$

$$= \max(0, 2.9) + \max(0, -3.9)$$

$N \leftarrow i=1$

$$L = (2.9 + 0 + 12.9)/3$$

约 = 5.27

最后对 L_i 求和做平均，得到结果 L 。

如果我们将第二个对象的类分数增加，会对损失函数产生影响吗？答案是不会的，它的损失仍然是 0，对最后的均值不会产生影响。

损失函数的最大最小值可能是什么呢？最小值可能是 0，我们找到一个非常完美的 W 函数，可以使得所有对象都完美匹配正确的分类，那 W 的损失函数最小值完全有可能是 0。而最大值为无穷，我们可以通过上面的合页函数图像观察到，如果某个对象对他的正确分类的对象的得分是趋向于负无穷的，那么它的损失函数就是正无穷的。

当你初始化这些参数，并且重新训练模型，通常我们先使用一些很小的随机值来初始化 W ，你的分数的结果在训练的初期倾向于呈现较小的均匀分布的值。并且问题在于如果你所有的 S 也就是说你所有的分数都近乎为 0 并且差不多相等，那你使用多分类 SVM 时，损失函数的和预计会是如何呢？答案是分类的数量减去 1（去掉了正确分类的分数），如果我们有 C 个分类，那我们得到的总和是 $C-1$ 。这个结论实际上是有用的因为这是一个有用的调试策略，当你使用这些方法的时候，当你刚开始训练的时候，你应该想到你预期的损失函数该是多大，如果在刚开始训练的时候你的损失函数在第一次迭代的时候损失函数并不等于 $C-1$ ，这意味着你的程序应该有 bug，你得去检查一下你的程序。

如果我们将对于 SVM 的所有损失求和会发生什么（意思是，我们之前在求取 L_i 时是将 $j \neq y_i$ 的损失去掉的，而这次如果我们把它加进来，会发生什么呢？）？答案是损失函数会加一，因为 $j = y_i$ 时， $S_j - S_{y_i} + 1 = S_{y_i} - S_{y_i} + 1 = 1$ 。则总的 L_i 就会加 1。

如果我们求取 L_i 不再是相加而是求平均呢？答案是 L_i 不会发生改变。当我们选择数据集的时候，由于分类的数量需要提前确定，则整体求平均的 L_i 只是比相加的 L_i 缩小了一个倍数，但是他指代的相对意义并没有发生改变，我们对损失函数的确定值并不是很在意。、

Q6: What if we used

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

如果我们改变 L_i 的公式，在 \max 中加一个平方项呢？这样的操作会使得这成为完全不同的另一个分类算法。这里我们使用了一种非线性的方法改变了在好与坏之间的权衡，所以实际上我们计算了另外一种损失函数。

一个损失函数的意义就在于量化不同的错误到底有多坏，同时分类器会犯不止一种类型的错误，我们如何对分类器可能犯得不同类型的错误进行权衡？如果我们使用平方项的损失函数，这意味着一个非常非常不好的错误就会更加的不好，成平方的不好，我们并不希望这样被完全严重分错的结果。

Multiclass SVM Loss: Example code

网易云课堂

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

```
def L_i_vectorized(x, y, W):
    scores = W.dot(x)
    margins = np.maximum(0, scores - scores[y] + 1)
    margins[y] = 0
    loss_i = np.sum(margins)
    return loss_i
```

译者注：在英文字母里，W的意思是两个U，所以发音是double U。网易云课：
一般会连读，我们会听到的是 ‘dab lju’，所以这里是在开玩笑四个U
 $f(x, W) = Wx$

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

E.g. Suppose that we found a W such that L = 0.
Is this W unique?

No! 2W is also has L = 0!

具有损失为 0 的 W 是否是唯一的？答案是不是的。我们可以整数倍的增大或者减小 W，损失仍然是 0.

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

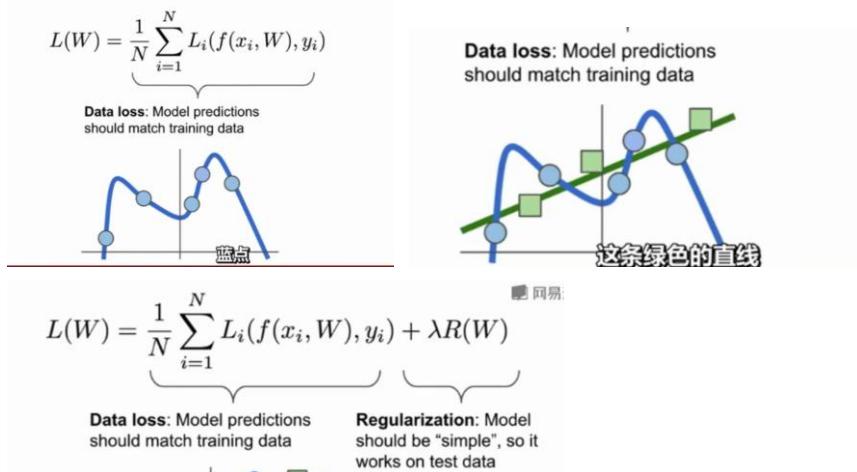
Before:

$$\begin{aligned} &= \max(0, 1.3 - 4.9 + 1) \\ &\quad + \max(0, 2.0 - 4.9 + 1) \\ &= \max(0, -2.6) + \max(0, -1.9) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

With W twice as large:

$$\begin{aligned} &= \max(0, 2.6 - 9.8 + 1) \\ &\quad + \max(0, 4.0 - 9.8 + 1) \\ &= \max(0, -6.2) + \max(0, -4.8) \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

既然有这么多 W，那么分类器应该如何选择 W 呢？我们仅仅是告诉分类器他需要尝试找到可以拟合训练集的 W。但是实际上，我们并不关心这很多关于拟合训练数据。机器学习的重点是我们使用训练数据来找到一些分类器，然后我们将这个东西应用于测试数据，所以我们并不关心训练集的表现，我们关心的是这个分类器在测试数据上的表现。所以，如果我们告诉分类器需要拟合数据，就有可能导致分类器可能会行为反常，所以这是一个具体的、典型的例子。假设我们有这样的数据集，就是图中的蓝点，我们拟合的可能是这条蓝色的曲线，而我们有可能更期望这条绿色的直线，而非这条完全适合所有的训练数据的蓝色曲线。这在机器学习领域内是一个核心的基础性问题，而我们通常的解决方式，则是这个正则化概念，所以这里我们需要为损失函数添加一个附加的项，除了数据丢失之外，除了高数分类器，需要拟合训练集之外，我们通常会添加一个项到损失函数内，这项被称为正则项，鼓励模型以某种方式选择更简约的 W，这里的简约取决于人物的规模和模型的种类，这里也体现了奥卡姆剃刀原理，也就是科学发现的规律，如果要让一个理论的应用更为广泛，也就是说，如果你有多个可以解释你观察结果的假设，一般来说，你应该选择最简约的，因为这可以在将来将其应用于解释新的观察结果，我们运用这种直觉的方式，基于这一思想并运用于机器学习中，我们会直接假设正则化惩罚项，这通常记为 R。这样一来，标准损失函数就有了两项，数据丢失项和正则项，这里有一些超参数 lambda 用于平衡这两项，这个参数 lambda 会是最重要的超参数之一。如果我们想要让回归的幂次降低，我们可以限制模型或者加入软惩罚来达到这一目的。



正在实践使用中的正则化有很多，最常见的可能是 L2 正则化或权值衰减，这个 L2 正则化就是欧式范数的权重向量 W 或有时平方规范或者有时候为二次范数，因为这会使你的推导变成现实更好一点。L2 正则化的理念实际上是对这个权重向量的欧式范数进行惩罚。正则化的目的主要是为了减轻模型的复杂度，而非去试图拟合数据。

Regularization

λ = regularization strength
(hyperparameter)

网易云课堂

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Max norm regularization (might see later)

Dropout (will see later)

Fancier: Batch normalization **最常见的是 L2 正则化** **stochastic depth**

L2 正则化如何判断模型的复杂度？

这里我们有一些训练的样本 x ，有两个我们正在考虑的不同的 W ，这里的 X 是一个包含四个 1 作为元素的四维向量，对于 w 我们来考虑两种不同的可能性。第一种情形是 w 的首元素为 1，其他三个元素都为 0；另一种情形 w 的每个元素都是 0.25。当我们线性分类时，我们真正讨论的是 x 与 w 的点乘结果，而在线性分类的语境下，这两个 W 其实是一样的，因为它们与 x 点乘结果是相同的。但是，如果我们观察这两个例子，哪种情形下的 L2 回归性会更好一些？因为第二种情况下的范数更小，因此它的回归性更好，因此 L2 度量可以用一种较为粗糙的方式去度量分类器的复杂性。我们需要记住，在线性分类器中， W s 代表的是 x 向量的值在多大程度上对输出有影响，所以 L2 正则化的作用是他更加能够传递出 x 中不同元素值的影响，它的鲁棒性可能更好一些，当你输入的 x s 存在变化的时候，我们的判断会铺展开来并且主要取决于 x 向量的整体情况，而非取决于 x 向量中的某个特定元素。而 L1 正则化则有完全相反的解释，实际上如果我们使用 L1 正则化，我们更加倾向于首元素为 1，其他为 0 的 W ，因为 L1 正则化对复杂性具有不同的概念，即那个模型可能具有较小的复杂

度。有可能在权重向量中，我们用数字 0 来描述模型的复杂度，所以问题是：我们如何度量复杂度，同时 L2 是如何度量复杂度的？它们视不同问题而定，针对你特殊的模型和数据以及不同的设置，你都需要仔细考虑，你需要思考在该任务中，复杂度应该如何度量。

L2 Regularization (Weight Decay)

网易云课堂

$$x = [1, 1, 1, 1]$$

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

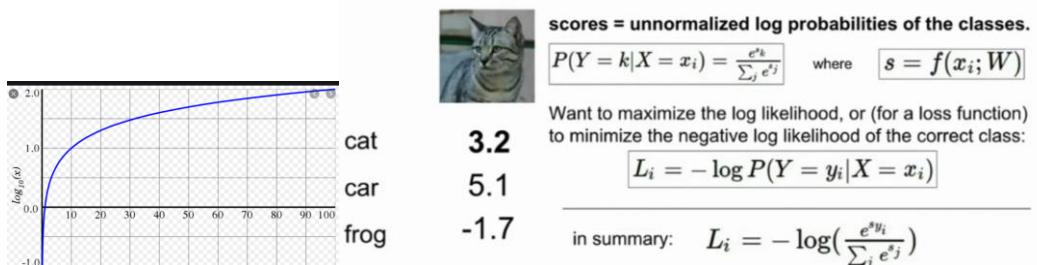
L1 背后的含义是：它通常更加喜欢稀疏一些，他倾向于让你大部分的 W 元素接近于 0，少数元素可以除外，他们可以被允许偏离 0。L1 度量复杂度的方式有可能是非零元素的个数，而 L2 更多靠的是 W 的整体分布，所有的元素具有较小的复杂性，所以具体选择哪个正则化，这个取决于你的数据以及具体的问题。

当面对单项和多项分类器时，我们应该做些什么？因为损失函数的使用，其实并不会对你正在做的有任何更改。增加正则化并不是为了改变假设的类别，他并不会更改线性分类器的工作方式，这里的关键点是：有可能多项式回归完全不是线性回归，它可以被视为线性回归，除了输入的是多项式延展项。另外，回归分类表明你并不允许用足够多的多项式系数。所以，我们可以这么思考，当你做多项式回归的时候，你可以写出一个多项式 $f(x)$ ，此时，你的 W s 就是这个多项式的系数，而对 W 进行惩罚可以强迫他朝着低阶多项式演进（减少多项式次数）。除非你在多项式回归分析中，你并不希望对系数进行参数化，而是希望对其他你想用的进行参数化，但通常的做法是你对模型中需要进行惩罚的参数排序，去强制它趋向于更加简单的假设。

另一个常用的损失函数是：多项逻辑斯蒂回归或称 softmax loss。

在使用多项 SVM 损失函数时，针对这些得分，我们并没有一个解释，当我们进行分类时，我们的模型函数 F ，给我们输出了 10 个数字，这些数字代表了我们的对象在这些类别的得分，针对多类别的 SVM 我们并没有对这些得分做过多的解释，我们只是说我们想要真实的分数，正确的分数最好比不正确的分数高，除此之外我们并没有解释这些得分的真正意涵。但是对于 softmax loss，我们将赋予这些得分一些额外的含义，并且我们会使用这些分数针对我们的类别去计算概率分布。所以当我们谈论分数时，我们将用这个所谓的 softmax 损失函数，我们将其指数化以便结果都是正数，接着我们利用这些指数的和来归一化它们。当我们将分数经过 softmax 函数处理过后，我们得到了概率分布，对于所有类别我们都有了相应概率，每个频率都介于 0 和 1 之间，所有类别的概率之和等于 1，这正是我们所期待的解释，我们拿这个概率和真是的概率分布进行比较。举一例，我们得知了我们的对象是猫的时候，我们就应该把所有的概率集中到猫的这个类别上去，所以我们可以得到猫的概率为 1，而对于其他类别而言概率为 0。现在我们需要做的就是去促使我们通过 softmax 得到的概率分布尽可能地去匹配上述地目标概率分布，即正确的类别应该具有几乎所有的概率，这里我们的损失函数就是：真实类别概率的对数再取负值。我们只要知道我们的期望概率趋近于 1，所以 log 函数是一个单调函数，大概趋势如下图，是一个单调递增的。找到 log 函数的最大值相对简单，我们只需要找到概率最大的就行了，因此我们选择了 log 函数。当我们针对正确类别最大化 $\log P$ ，意味着我们想要他变高，但是损失函数是用于度量坏的程度的函数，而非好的程度，所以我们加了一个负号，让他更加符合我们的预设。

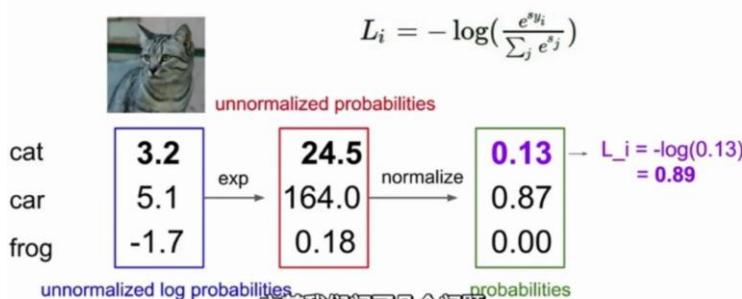
Softmax Classifier (Multinomial Logistic Regression) 网易云课堂



做一个总结，我们使用 softmax 对分数进行转化处理，并得到正确类别的损失函数是 $-\log P$ 。

举一例：

Softmax Classifier (Multinomial Logistic Regression) 网易云课堂

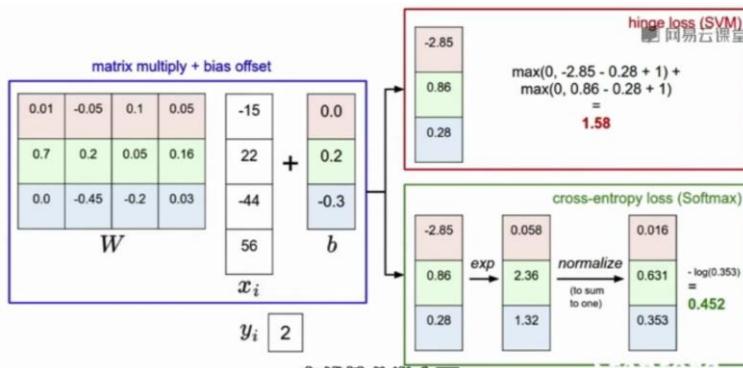


我们首先通过线性分类器得到各类别的分数，这时区别于 SVM 损失函数，我们将其做处理之后在应用到 softmax 损失函数，我们首先对其取对数，保证每个分数处理后都为正数，再对其进行归一化处理，在将其取对数(这里的 log 其实就是 lg)后取负数，

Softmax 损失函数的最小和最大值分别是多少？最小值 0，最大值为无穷大。如果我们的工作十分顺利，那我们得到正确概率应该是 1，那么 lg1 等于 0，取负值仍然为 0，那我们得到的最小值就是 0 了。而十分错误的类型的概率为 0，而根据 log 的图像，我们知道，在 $x \rightarrow 0$ 时， \log 趋向于负无穷。而由于计算机限制，我们永远不会得到这个极值的精度。

当我们的得分 S 全都趋向于 0 时，我们得到的损失是？ $\lg(c)$ 。在第一次迭代时，如果他不是 c 取自然对数，那么就出现了问题。

我们对这两种损失函数做出比较：



在线性分类方面，我们将 W 和输入向量相乘，得到分值向量。那么，两个损失函数的区别是我们如何来解释这些分值，进而量化度量到底有多坏，所以对于 SVM，深入研究并观察，正确的分类的分值和不正确分类的分值的边际。而对于 softmax 或者交叉熵损失函数，我们打算去计算一个概率分布，然后查看负对数概率。

Softmax vs. SVM

网易云课堂

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \quad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $y_i = 0$

Q: Suppose I take a datapoint and I jiggle a bit (changing its score slightly). What happens to the loss in both cases?

假设我们有三个分数，忽略底部的

部分，针对SVM损失函数，我们对正确分类的分数稍微修改也不会影响损失函数，因为在SVM中，正确的分类的损失函数就是0。但是在softmax中情况很不同，softmax的目标是将概率质量函数(离散分布值)等于1，如果我们将正确分类的分数提高，错误分类的分数减小，那么正确的分类仍然会朝着正无穷扩张，而错误的分类则会朝着负无穷扩张。两者的区别在于：SVM中，我们得到了一个正确分类的分数，就会将它丢弃，不再关心这个被丢弃的数据点，而在softmax中，总是试图不断提高，每个数据点都会越来越好。

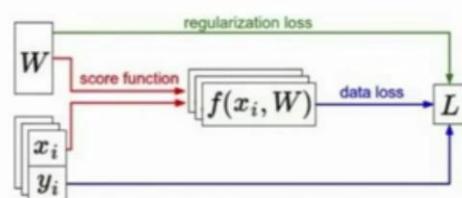
我们有一些 xs 和 ys 的数据集，我们使用我们的线性分类器来获得一些分数函数，然后我们使用SVM或者softmax损失函数来判断我们的预测好坏与否，最后我们添加一个正则惩罚模型的复杂性防止过拟合，来选择我们想要的模型。当我们完成这些步骤，找到了我们想要的 W ，在最小化了这个最终的损失函数。

Recap

网易云课堂

- We have some dataset of (x, y)
- We have a **score function**: $s = f(x; W) = Wx$ e.g.
- We have a **loss function**:

$$\begin{aligned} & \text{Softmax} \\ L_i &= -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \\ & \text{SVM} \\ L_i &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \\ L &= \frac{1}{N} \sum_{i=1}^N L_i + R(W) \quad \text{Full loss} \end{aligned}$$



所以，我们应该如何去执行这个步骤？我们如何才能发现这个 W 使损失最小化？

优化

在现实中，我们基本不可能一下子找到这个能使损失最小化的 W ，因此我们倾向于从某个解决方案开始，使用多种不同的迭代方法，然后逐步对他进行改进。

有可能我们首先想到随机搜寻，我们需要很多的权重值，随机采样，然后将他们输入损失函数，再看看效果究竟如何。我们有可能通过随机搜索来训练一个线性分类器，

Strategy #1: A first very bad idea solution: Random search

网易云

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
    loss = L(X_train, Y_train, W) # get the loss over the entire training set
    if loss < bestloss: # keep track of the best solution
        bestloss = loss
        bestW = W
    print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues)
```

再看看它们效果如何

使用斜率，所谓斜率就是函数的导数，所以如果我们有一个一维函数 f ，它将 X 作为标量，输出为曲线的高度，我们就能计算出任何想象中的一点的斜率，如果我们向任意方向前进一步 h ，然后比较这一步前后的函数值的差别，然后将步长趋向于 0 的时候，我们就能得到那一点的函数的斜率，而且这个概念可以很简单的推广到多元函数中，所以在实际应用中， x 可能不是标量，而是整个向量，所以我们需要有多元（多参数）这一概念，在多元情况下生成的导数，就叫做梯度。梯度就是偏导数组成的向量，梯度有和 x 一样的形状，梯度中的每个元素可以告诉我们，相关方向上函数 f 的斜率，所以梯度就成为偏导数的向量，它指向函数增加最快的方向，相应的，负偏导数的方向就指向了函数下降最快的方向。概括起来，这个地形任意方向的斜率，他就等于这一点上梯度和该点单位方向的向量的点积。梯度非常重要，它给出了函数在当前点的一阶线性逼近。因此在实践中，很多深度学习都是在计算梯度，然后用这些梯度迭代更新你的参数向量。在

Strategy #2: Follow the slope

网易云课堂

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

计算机中计算梯度的一个简单方法是使用有限差分法，这就回到了梯度的极限定义。我们有当前的 W 的损失函数结果，我们想要知道每改动 W 的每个元素会对损失函数产生什么样的影响，这里我们先对 W 的第一个元素加一个很小的 h 值，发现损失函数发生了一点改变，

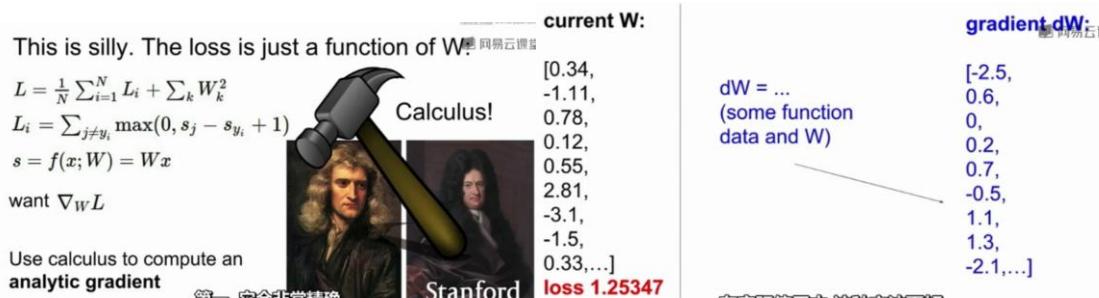
然后我们可以使用这个有限差分在梯度的第一维实现有限差分逼近，计算出梯度的近似值。

current W:	W + h (first dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...] loss 1.25347	[0.34 + 0.0001, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...] loss 1.25322	[?, ?, ?, ?, ?, ?, ?, ?, ?,...]



我们将 W 的第一维恢复，再在第二维方向上增加一小步，再对第二维计算梯度。不断重复这样的过程，最终可以得到整个梯度向量。

但是如果数据很大，W 长度很长，那么整个计算过程就会非常缓慢，效果也不会好，因此我们在实际中不会使用有限差分计算。



In summary:

网易

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check**.

尽管数值梯度很好理解，但是我们在实

际中总是使用解析梯度。

而数值梯度是一个很好的 debug 工具，当我们使用解析梯度时，我们可以使用数值梯度来单元验证，从而确定我们输入的解析表达式是正确的。

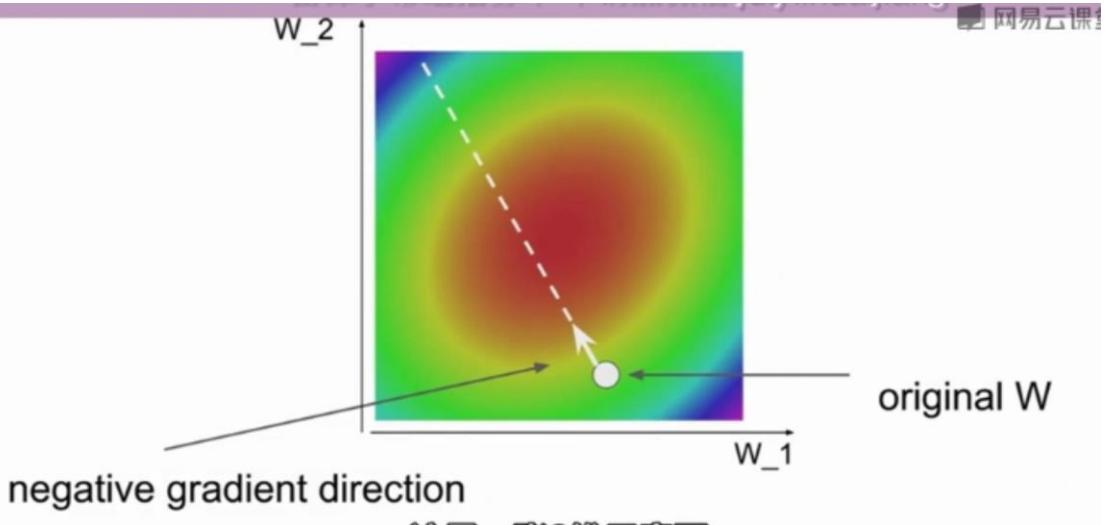
梯度下降算法：

```
# Vanilla Gradient Descent

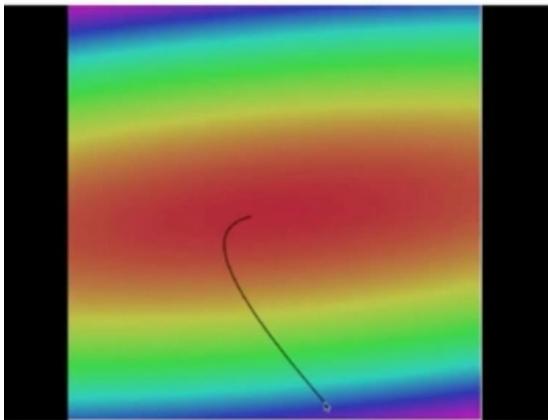
while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```

在梯度下降算法中，首先我们初始化 W 为随机值，当为真时，我们计算损失和梯度，然后向梯度相反的方法更新权重值。因为，你需要记住，梯度是指向函数的最大增加方向，所以梯度减小方向应该是指向函数的减小方向。因此，我们向梯度减小地方向不断地迭代，然后一直重复，就可以使得网络收敛。但是步长是一个超参数，这就告诉我们每次计算梯度时，这就告诉我们每次计算梯度时，在那个方向前进多少距离，这个步长也被叫做学习率。它可能是你需要设定地最重要的一个超参数。决定学习率或者步长，是我们首先要做的事情。

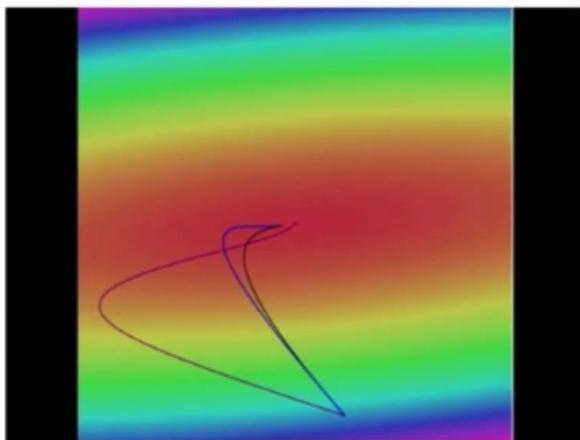
下图中的中心区域是误差较小的区域，而边界处则代表了较高的误差值，是我们需要避免的。



逼近中心点的例子



更高级的迭代例子



总的误差我们取平均值，但是在实际中， N 有可能会非常非常大，我们要不断地计算损失函数，因此占用资源会非常大，计算非常慢。因为梯度是一个线性运算符，当你打算计算误差函数的梯度值时，你会发现，误差函数的梯度值是每个单项误差梯度值的总和。所以，如果我们要再次计算梯度，我们似乎就需要迭代整个训练数据集，迭代所有 N 个样本，计算非常缓慢。因此我们在实际中采用随机梯度下降，他并非是计算整个训练集的误差和梯度值，而是在每一次迭代中，选取一小部分训练样本成为 minibatch(小批量)。按惯例，这里都取 2 的 n 次幂，然后我们使用这个 minibatch 来估算误差总和以及实际梯度。代码如下所示

Stochastic Gradient Descent (SGD)

网易云课堂

Full sum expensive
when N is large!

Approximate sum
using a **minibatch** of
examples
32 / 64 / 128 common

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$
$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

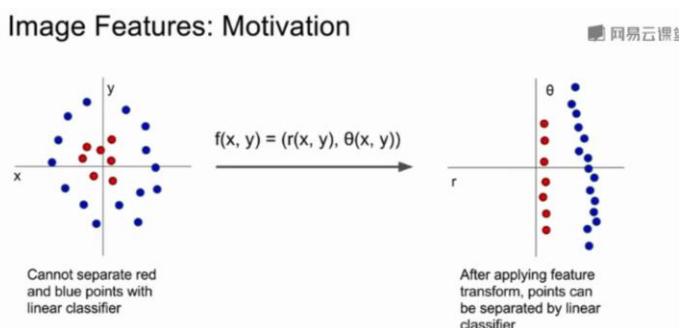
```
# Vanilla Minibatch Gradient Descent

while True:
    data_batch = sample_training_data(data, 256) # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # perform parameter update
```

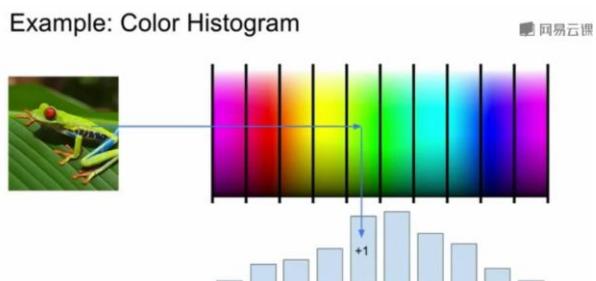
图像的特征：

在前面提到过，直接通过特征来卡分类不是个好选择，会受到各种因素的影响，将其传递给线性分类器的表现并不理想。所以，当深度神经网络大规模应用前，常见的方式就是采用两步走策略：(1) 首先，拿到图片，计算图片的各种特征表示，比如有可能计算于图片形象相关的数值，然后将不同的特征向量合在一起，得到图像的特征表述 (2) 然后，这些特征表示会作为输入源传入线性分类器，而非原始的像素。

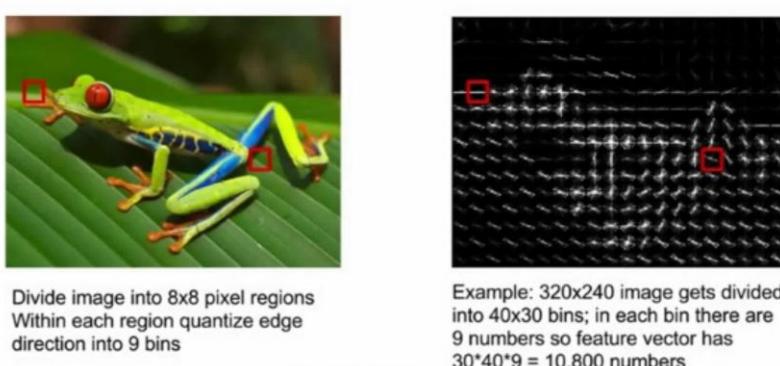
我们有一个训练数据集如下图左侧所示，红点分布在中间，蓝点散布在其周围。而对于这种数据，我们不可能一个线性决策边界来将红点和蓝点分开。但如果我们使用灵活的特征转换，比如这里我们采用极坐标转换，现在我们得到转换特征，就可以将复杂的数据点变成线性可分的了，就可以通过线性分类器进行正确的分类了。



极坐标转换有可能对图片的作用不大，但我们可以选择对图片有意义的特征转换，比如，颜色直方图：获取每个像素所对应的光谱，将其分到柱状内，将每一个像素都映射到这些柱状中，然后计算出每个不同的柱状中像素出现的频次，这个转换从全局上告诉我们图像中有哪些颜色。



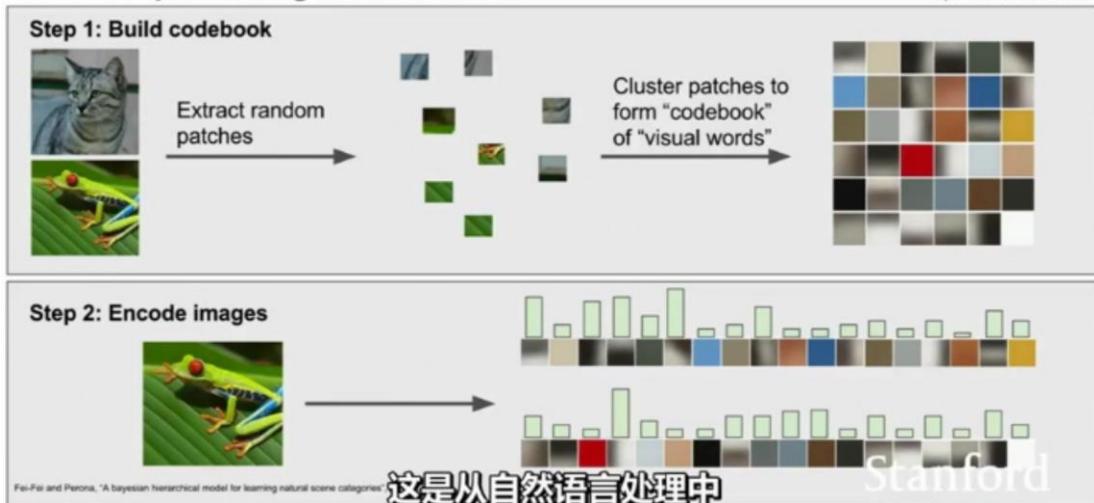
再比如，方向梯度直方图：首先获取图像，然后将图像按八个像素分为八份，然后在八个像素区的每一个部分计算每个像素值的主要边缘方向，把这些边缘方向量化到几个组然后在每一个区域内计算不同的边缘方向从而得到一个直方图。现在，你的全特征分量就是这些不同组的边缘方向直方图，这个直方图是从图像的八个区域得来。



语袋:

Example: Bag of Words

网易云课堂



反向传播和神经网络

$$s = f(x; W) = Wx$$

scores function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

SVM loss

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

data loss + regularization

want $\nabla_W L$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Numerical gradient: slow :, approximate :, easy to write :)

Analytic gradient: fast :, exact :, error-prone :(

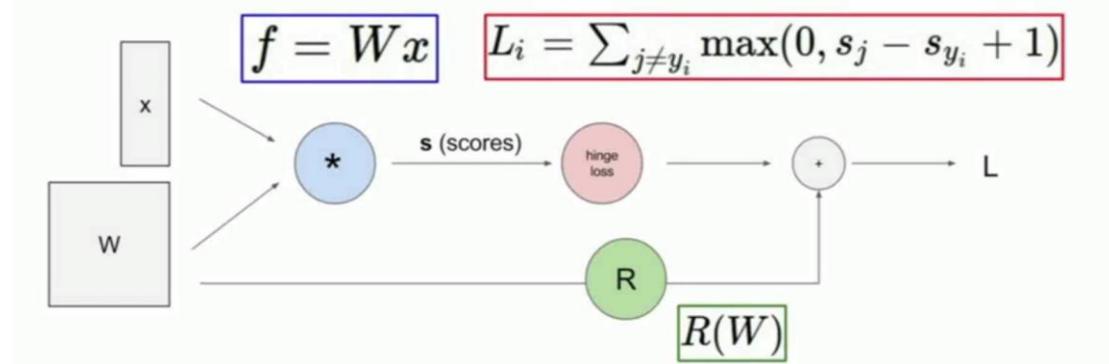
In practice: Derive analytic gradient, check your implementation with numerical gradient

如何计算任意复杂函数的解析梯度，要用到一个叫做计算图的框架。大体上说，计算图就是我们用这类图来表示任意函数，其中图的节点表示我们要执行的每一步计算。

举一例，如我们学习过的线性分类器，输入是 x 和 W ，这个乘的节点代表矩阵乘法，是参数 W 和数据 x 的乘积输出的得分向量，然后我们有一个节点代表 hinge loss 来计算数据损失项 L_i ，我们还有一个正则表达式 $R(W)$ 在右下角，这个节点计算了正则项，在最后的总的损失 L ，是正则项和数据项的和，这里的好处是，一旦我们能用计算图来表示一个函数，我们就能用所谓的反向传播技术递归地调用链式法则来计算图中的每个变量的梯度。

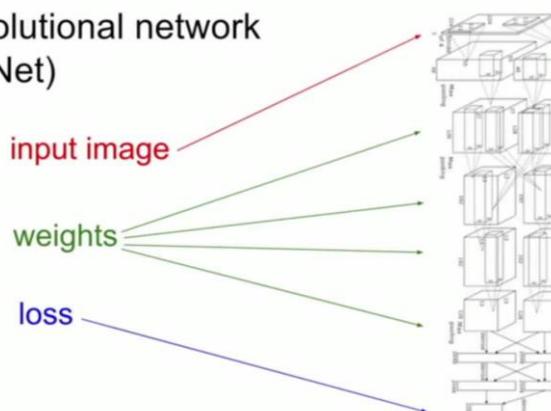
Computational graphs

网易云课堂



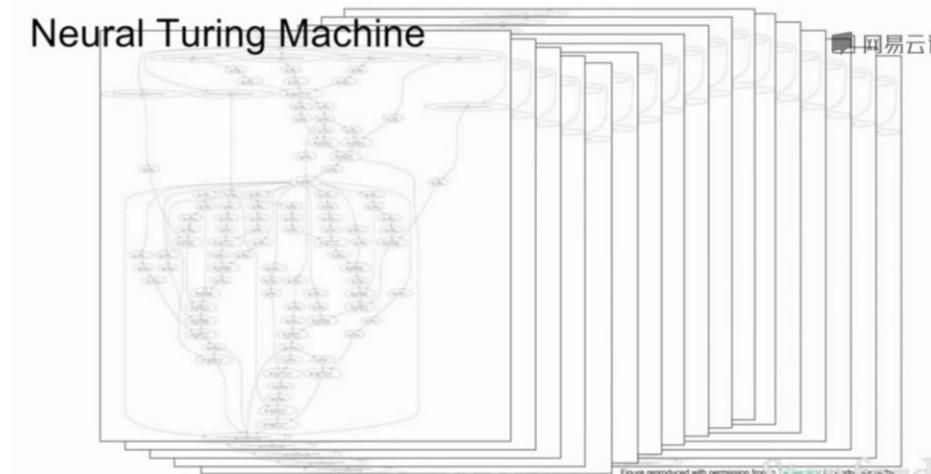
神经网络

Convolutional network
(AlexNet)



神经图灵机器

Neural Turing Machine



反向传播的工作流程：

这里我们有一个很简单的函数，而我们要找到函数输出对应任意变量的梯度，

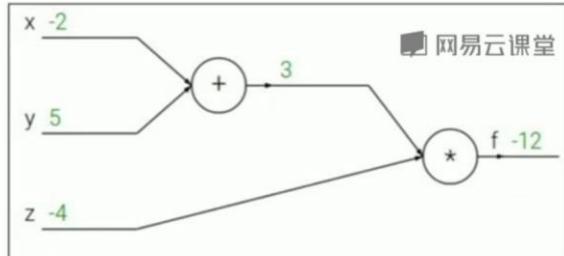
第一步：就是用计算图来表示我们的整个函数

第二步：就是要做这个网络的前向传播，给定了每个变量对应的值，如下图所示，我们将这些值写到计算图中，然后计算得到中间值，

Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$



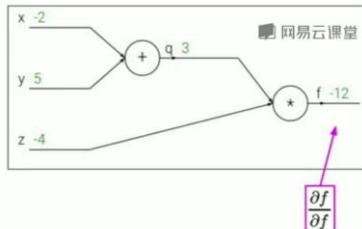
在这里我们想给每个中间变量一个名字，我们把做完加法的中间变量，叫做 q ，这里 $q=x+y$ ；然后是节点 $f=q*z$ 用到了中间节点 q ，我们把梯度也写在这里， q 所对应的 x 和 y 的梯度都是 1，因为是相加。而 f 在 q 和 z 方向上的梯度分别是 z 和 q ，因为是乘法运算。

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

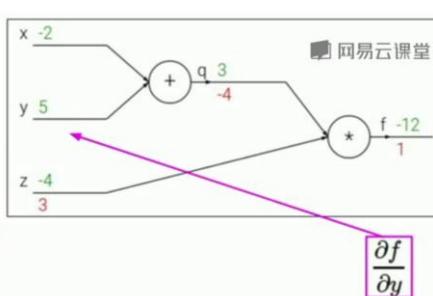
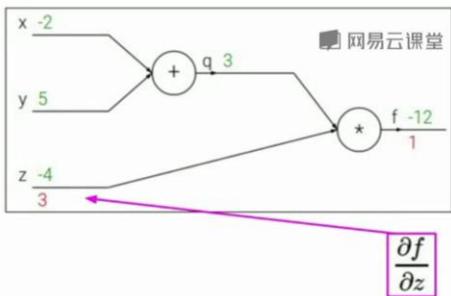
$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

我们想要知道的是 f 对于 x, y, z 的梯度。反向传播是链式法则的递归调用，我们先计算图的最后面，我们从后往前计算所有的梯度，这里我们从右边的图末开始，我们要计算输出对最后一个变量 f 的梯度，也就是 f ，如下图所示，很明显的，这个梯度是 1

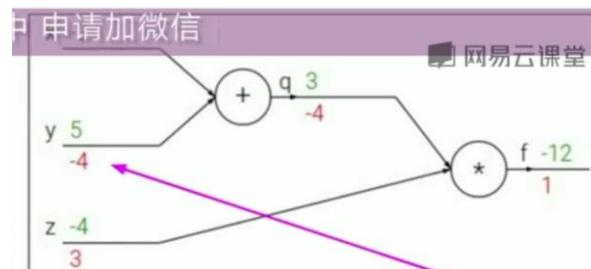


然后，我们从后往前，想要计算在 z 方向上的梯度，我们知道 df/dz 等于 q ，而 q 的值为 3，所以我们知道了 df/dz 等于 3。如果我们想要接着计算 $df/dq=z=-4$



现在我们继续沿着计算图从后往前计算，我们想要找到 df/dy ，但在这种情况下关于 y 的梯度不能直接得到， y 跟 f 并没有直接关系，但是通过节点 q 来连接，我们可以通过链式法则，

通过相对的 $(df/dq) \cdot (dq/dy)$ 来求得 y 的梯度, 而 df/dq 是我们已知的 $= -4$, 而 $dq/dy = 1$, 如果我们稍微的改动 y , 则 q 就会发生变化, 因此这就是 y 对 q 的影响 (dq/dy) 。求解得到 $df/dy = -4$ 。

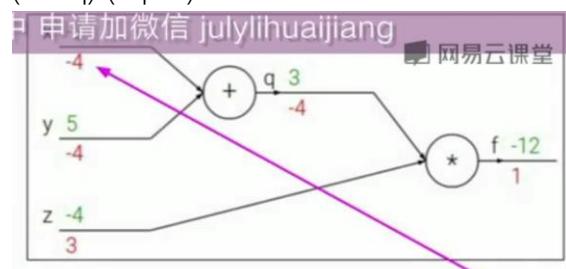


Chain rule:

$$\frac{\partial f}{\partial y}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

同样的, 如果我们想要求解 x 方向上的梯度, 我们可以用相同的方法, 则结果为 $(df/dq) \cdot (dq/dx) = -4$

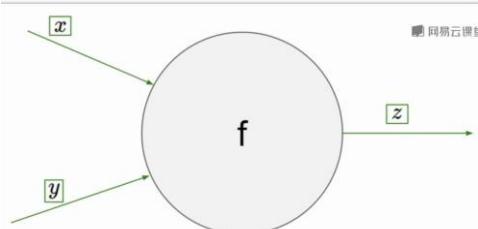


Chain rule:

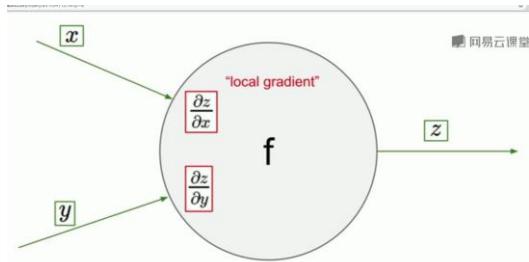
$$\frac{\partial f}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

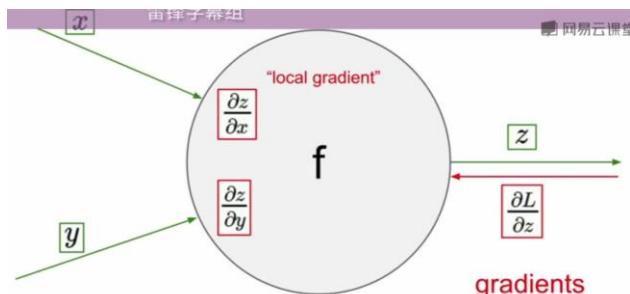
在计算图中, 每个节点只知道与它直接相连接的节点, 所以在每个节点上, 我们有连接这个节点的本地输入, 也就是流入这个节点的值, 然后我们也有这个节点直接的输出, 这里我们本地输入的是 x 和 y , 输出的是 z ,



我们也知道这个节点的梯度, 我们可以计算出关于 z 在 x 方向上的梯度 (dz/dx) 和 z 在 y 方向上的梯度 (dz/dy) 。



对于每个节点基本都能得到它的本地输入和直接传递到下一个节点的输出，相对于节点的输入，我们也可以计算出他们相应的梯度是节点的直接输出。

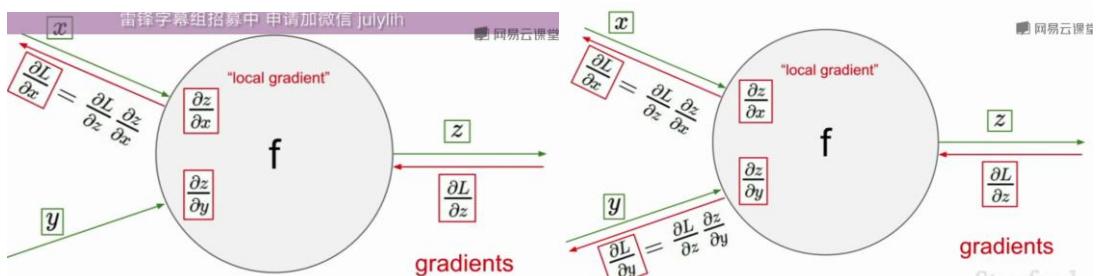


我们从最后的地方一直到开始的地方，

当我们到达每一个节点时，每一个节点都会得到一个从上游返回的梯度，这个梯度时对这个节点的输出的求导。等到我们在反向传播中到达这个节点，我们就计算出来了最终损失 L，下面我们想要找到关于节点的输入的梯度，即在 x 和 y 方向上的梯度。

我们用链式法则来计算得到 x 对于 L 的梯度，在链式法则中，我们通常把上游的梯度值传回来，再乘以本地的梯度值，从而得到了关于输入的梯度值。

那这个方法是否是仅在这个函数的当前取值时有效？给函数输入一个当前值（临时值），我们可以写出它的表达式，而且是关于变量的表达式，我们可以对 L 关于 z 进行求导，我们以 L 对 z 求导可以得到一个表达式，而 z 对 x 求导会得到另外的表达式，现在，我们把这些数的值，现在我们将这些表达式输入进去，就可以得到关于 x 的梯度值，如果你递归地将表达式插入到所有的表达式，那么对于 x 的梯度，我们会有一个很简洁的表达式。

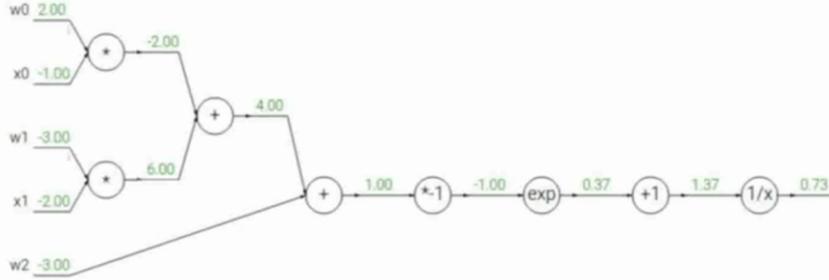


主要的工作在于要在每个节点上计算我们所需的本地梯度，然后跟踪这个梯度，在反向传播过程中，我们接受从上游传回来的这个梯度值，我们用这个梯度值来乘以本地的梯度值，就可以得到我们想要传回连接点的值，在下一个节点进行反向传播时，我们不考虑除了直接相连的节点之外的任何东西。

举一例，我们将下面的函数转换为一个计算图，在这个计算图中，

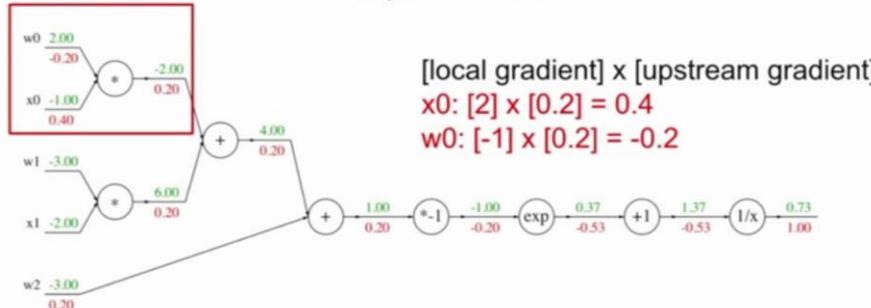
Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$

网易云



Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$

网易云课堂



$$\begin{array}{lll} f(x) = e^x & \rightarrow & \frac{df}{dx} = e^x \\ f_a(x) = ax & \rightarrow & \frac{df}{dx} = a \end{array} \quad \left| \quad \begin{array}{lll} f(x) = \frac{1}{x} & \rightarrow & \frac{df}{dx} = -1/x^2 \\ f_c(x) = c + x & \rightarrow & \frac{df}{dx} = 1 \end{array} \right.$$

相对于直接推到表达式的梯度表达式，为什么这样计算会更简单一点？

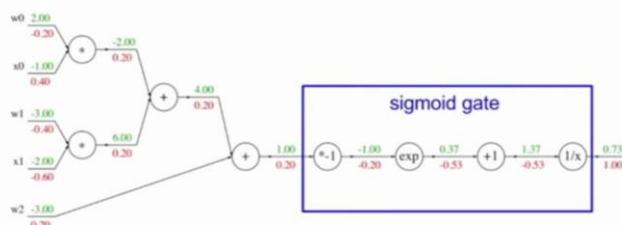
我们所有处理的本地表达式，我们要做的就是填充每个值，然后使用链式法则，从后往前乘以这些值，得到对所有变量的梯度，从后往前乘以这些值，得到对所有变量的梯度。

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

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

sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

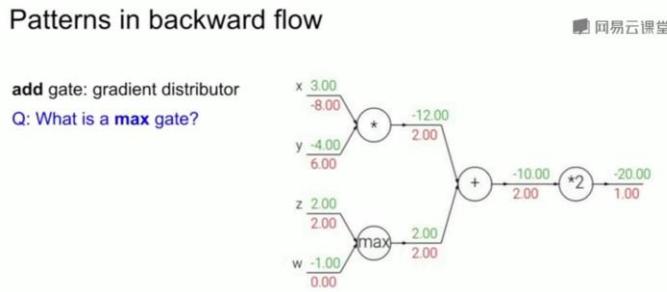


因此这里，我们也可以用相同的方法得到在 w_1 和 x_1 方向上的梯度，当我们在创建这些计算图的时候，我们可以以我们想要的任意间隔尺寸来定义计算节点，所以在这种情况下，我们把它分解成我们能够达到的最简单的形式，分解成加法和乘法运算。但是实际上，我们可以把一个个节点组合在一起形成更复杂的节点，比如上图中的 sigmoid 节点，我们可以聚

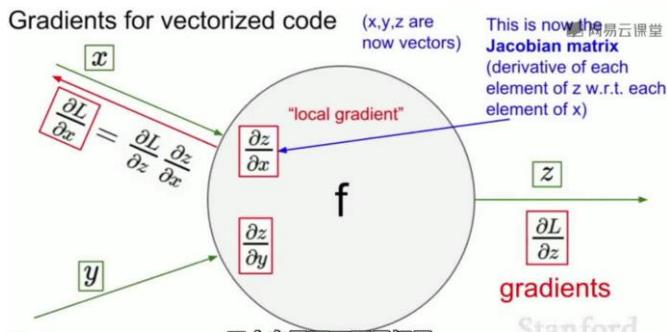
合想要集中处理的节点，从而去组成一些在某种程度上稍微复杂的节点，只要你能够写出他的本地梯度。

Max 门 (Max gate)：取输入的最大值作为输出。而对于 max gate 的 z 梯度和 w 梯度分别为 2 和 0，max 会获取上游传递过来的梯度，并且将其传递给下游的最大值输入，其他的输入梯度均为 0。我们可以看到，只有输入的最大值才能传递到计算图的剩余部分，也就是说其他输入对计算的影响程度为 0，而这个最大值的影响为 1，而这个影响程度也就是我们所谓的梯度。

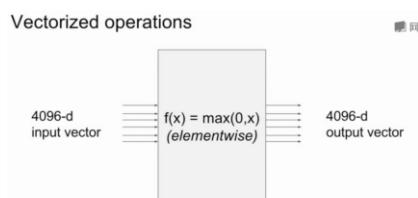
Mul Gate (乘法门)：其输入的本地梯度基本上就是另外一个变量乘以上游梯度



来看一下高维情况下的反向传播，此时，我们有三个输入 x,y,z，并且三个向量不再是标量，这时整个计算流程还是一样的，唯一不同的就是我们的链式法则变成了雅可比矩阵



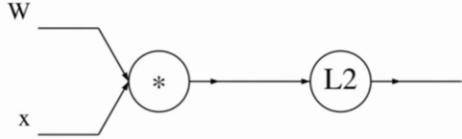
来看一个例子，在这里我们的输入是 4096 个元素的向量，通过这个 f 是取向量和 0 之间的最大值，最后得到一个 4096 个元素的输出。那么这个雅可比矩阵的规模就是 4096×4096 。而在实际中由于我们要进行批处理，要处理的矩阵规模会大得多，比如 409600×409600 。这个数据规模过大，因此在实际操作中多数情况下不需要计算这么大的雅可比矩阵，在这个 f 中，我们的输出只和对应的输入有关系，因此在求每个元素求取偏导数时，只有一个对应的输入元素的偏导数是非零的，这样的话，整个雅可比矩阵实际上是一个对角矩阵。因此我们不需要将整个矩阵写出来，而我们只需要输出向量关于 x 的偏导，然后将结果作为梯度填进雅可比矩阵。



再看一例，其中 x 为 n 维向量， W 为 n 维的矩阵

$$\forall \text{vectorized example: } \|x^T W\|_3 = \|W \cdot x\|_3 = \sum_{i=1}^n (W^T x)_i^{\frac{1}{3}}$$

先写出计算图



我们给出一个输入，如图所示，并且得出计算结果

A vectorized example: $f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^n (W_i \cdot x_i)^2$

$$W = \begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{bmatrix}$$

$$x = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \dots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \dots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = \|q\|^2 = q_1^2 + \dots + q_n^2$$

$$\begin{bmatrix} 0.22 \\ 0.26 \end{bmatrix} \xrightarrow{*} \begin{bmatrix} 0.116 \\ 1.00 \end{bmatrix}$$

然后我们开始进行反向传播，我们需要得到的是 q 对于 f 的影响，根据公式，我们用 f 对于 q_i 的求导的结果恰好是两倍的 $q_i(2q_i)$ 。

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\nabla_q f = 2q$$

$$\begin{bmatrix} 0.22 \\ 0.26 \\ 0.44 \\ 0.52 \end{bmatrix} \xrightarrow{*} \begin{bmatrix} 0.116 \\ 1.00 \end{bmatrix}$$

所以向量的梯度总是和原向量保持相同的大小，每个梯度的元素代表着这个特定元素对最终函数影响的大小。

所以，对于 W 的梯度到底代表着什么？我们来观察 q ， q 的每个元素对应着 w 的每个元素，这就是雅可比矩阵。我们 q 中的第一个元素是 q_1, W 中的第一个元素对应着 $W_{1,1}$ ，所以 q_1 是对应着 $W_{1,1}$ 吗？

$$\frac{\partial q_k}{\partial W_{i,j}} = \mathbf{1}_{k=i} x_j$$

我们将这个结论推广一下，关于 w_{ij} 的 q_k 梯度等于 x_j 。

$$\begin{aligned} \frac{\partial f}{\partial W_{i,j}} &= \sum_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} \\ &= \sum_k (2q_k)(\mathbf{1}_{k=i} x_j) \\ &= 2q_i x_j \end{aligned}$$

再通过链式法则，取乘法就能得到了 f 对于 W

每个元素的梯度，最终求和就可以得到最后的梯度。

A vectorized example: $f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^n (W_i \cdot x_i)^2$

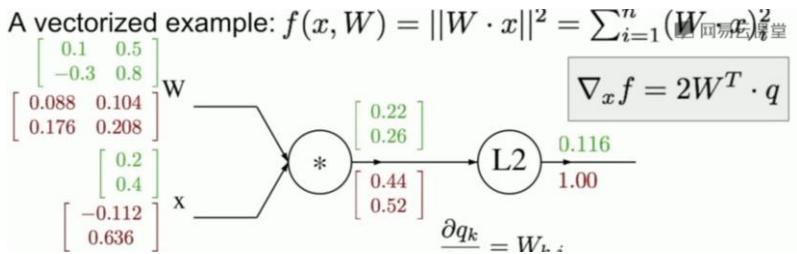
$$W = \begin{bmatrix} 0.1 & 0.5 \\ -0.3 & 0.8 \\ 0.088 & 0.104 \\ 0.176 & 0.208 \end{bmatrix}$$

$$\nabla_W f = 2q \cdot x^T$$

根据得到的梯度，我们推导出其关系。

而一件很重要的事是检查变量梯度的向量大小应该和变量向量大小一致，因为每个梯度向量都量化了每个元素对最终输出影响的贡献。

然后我们再做同样的操作得到 x 的梯度。



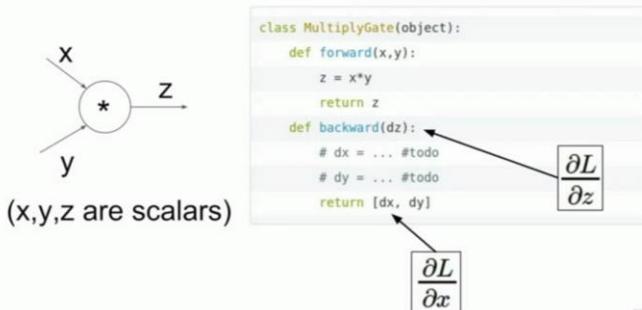
在此时，得到的雅可比矩阵和 W 一致，在这里雅可比矩阵的维数并不大

Graph (or Net) object (rough psuedo code)

```
class ComputationalGraph(object):
    ...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

这种模式像是模块化计算，在前向传播的 API 中，我们计算节点输出，而在反向传播的 API 中，我们计算节点梯度。

来看一下偏导数们的运算，在前向传播中，我们计算得到 $x \cdot y$ 的值，即 muli gate 的计算结果，然后我们在反向传播中输入 dz ，然后返回 dx 和 dy 的值



因为是 muli gate，则 x 的本地节点梯度就是 y 的输入值，而 y 的本地梯度就是 x 的输入值

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

总结：当你使用神经网络，它们都将会非常庞大和复杂，将所有参数的梯度公式写下来是不现实的，所以为了得到这些梯度，我们使用了反向传播和链式原则，计算出相对于所有中间变量的梯度，这些中间变量是你的输入、参数和中间所有的值

Summary so far...



- neural nets will be very large: impractical to write down gradient formula by hand for all parameters
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the **forward()** / **backward()** API
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

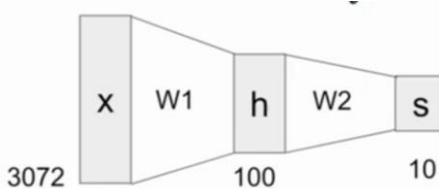
神经网络

此前我们使用了许多线性的函数来评分，我们将其作为需要优化的函数来使用

(Before) Linear score function: $f = Wx$

如果我们想还用这种形式，在神经网络中也应用这些评分函数，

(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

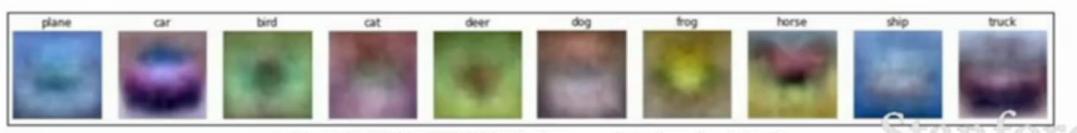


我们先通过 x 和 W_1 得到中间结果，然后再取 0

和中间结果的（线性层的）最大值与 W_2 得到评分，这些非线性计算非常重要，因为如果你只是在顶层将线性层和在一起，他们将会被简化为一个线性函数，所以我们首先有线性层，然后有这个非线性计算，继而在顶层再加入另外一个线性层，然后到这里，我们可以得到一个计分函数用于输出计分向量。

一般来说，所谓神经网络就是由简单函数构成的一组函数，在顶层堆叠在一起，我们用一种层次化的方式将它们堆叠起来，为了形成一个更复杂的非线性函数，所以这就是基本的多阶段分层计算。

在前面学习的 W 中的每一行都是一个类的模板，例如车这一类。我们输入的样本可能跟 W 的模板不同，例如这是个黄色的车，然后我们用 W_1 来为其评分，得到评分后的分数--- h ，然后我们利用 W_2 为不同分类的分数进行加权，最终得到我们想要的得分函数。其中非线性函数一般在 h 之前



利用这个原理，我们可以得到任意深度的神经网络

(Before) Linear score function: $f = Wx$

(Now) 2-layer Neural Network or 3-layer Neural Network

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

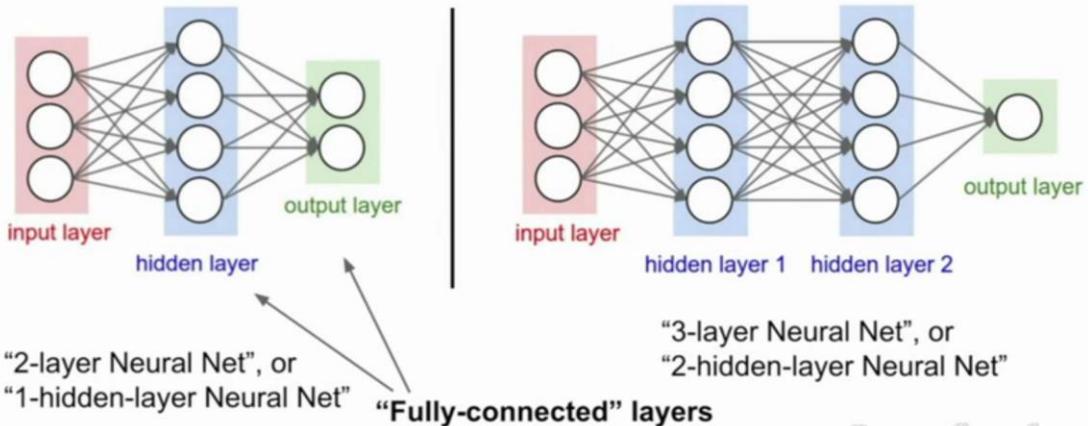
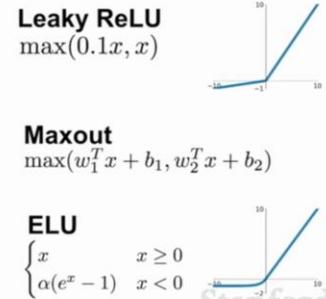
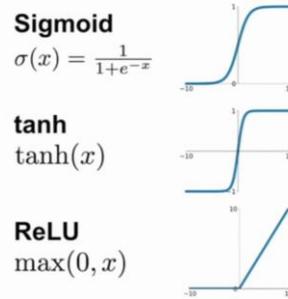
```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
    h = 1 / (1 + np.exp(-x.dot(w1)))
    y_pred = h.dot(w2)
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h.T.dot(grad_y_pred)
    grad_h = grad_y_pred.dot(w2.T)
    grad_w1 = x.T.dot(grad_h * h * (1 - h))

    w1 -= 1e-4 * grad_w1
    w2 -= 1e-4 * grad_w2
```



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

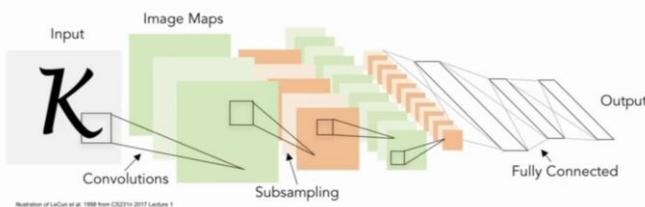
Summary

网易云课堂

- We arrange neurons into fully-connected layers
- The abstraction of a **layer** has the nice property that it allows us to use efficient vectorized code (e.g. matrix multiplies)
- Neural networks are not really *neural*
- Next time: Convolutional Neural Networks

卷积神经网络历史

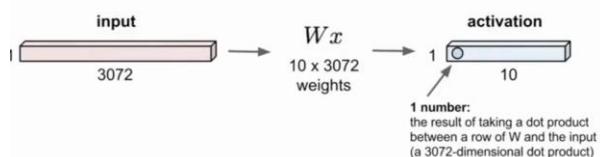
与普通的神经网络类似，但是需要训练卷积层，因为卷积层更能够保留输入的空间结构



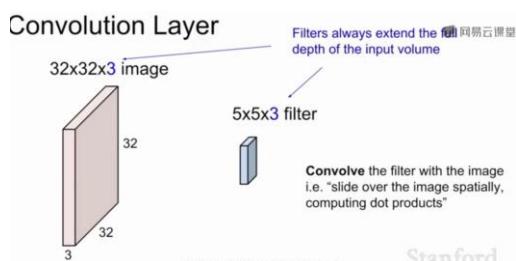
卷积和池化

我们将一张三位图片展开为 3072×1 的向量

$32 \times 32 \times 3$ image \rightarrow stretch to 3072×1

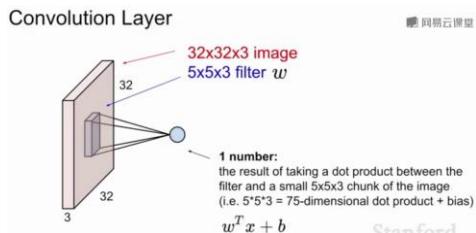


得到一个数值，这个数字也就是该神经元的值，在上面的情况下，我们有十个神经元的值。与全连接层不同，卷积层主要差别是可以保留空间结构，使用我们原来的图片空间结构 $32 \times 32 \times 3$ 而非展开成 3072×1 的长向量，从而使得我们可以保持图片的结构。我们的权重是一些小的卷积核，例如下面的 $5 \times 5 \times 3$ 大小的，我们将这个卷积层在整个图像中滑动，计算出每个空间定位时的点积结果。



首先，我们采用的卷积核总是会将输入量拓展至完全，所以他们都是很小的一个空间区域，但它们通常会遍历所有通道。

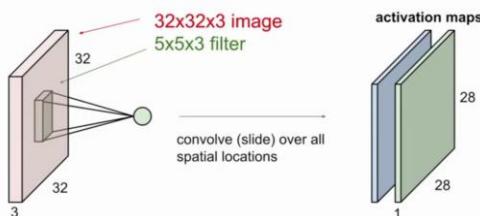
然后我们采用这个卷积核，并且给定一个空间区域，在这个卷积核和图像块间，进行点积运算，我们要做的就是在图像空间区域中覆盖这个卷积核，然后进行点积运算，也就是将卷积核每个位置元素和与之相对应的图像区域的像素值相乘，这个区域是从图像上取出来的，经过运算后会给我们一个点积的结果，在这个例子中，我们进行了 $5 \times 5 \times 3$ 次运算，这是乘法运算的次数，之后我们再加上偏置项，这就是卷积核 W 的基本方法就是用 W 的转置乘以 X 再加上偏置项。我们可以这么理解，这里的卷积核 W 并非一个 $5 \times 5 \times 3$ ，而是就是一个函数有 75 个元素的向量，当我们将 W 对 X 进行变换而进行点乘时，我们首先要做的就是将其展开，在我们进行点积之前。



我们如何滑动卷积核并且遍历所有空间位置呢？

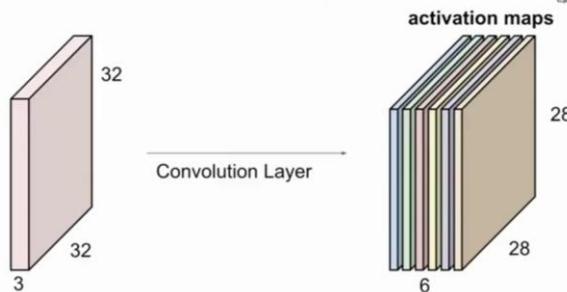
我们从左上角开始滑动卷积核，这里滑动的步长可以是一个像素点也可以是两个像素点，这是可以选择的，而这个选择会影响我们得到的输出，但一般情况下我们会按照栅格形式来进行滑移。

我们采用了一卷积核，然后将他在图像的整个平面上进行滑移，然后我们输出它的激活映射，它里面的值就是卷积核在每个位置求得的结果，当我们在处理一个卷积层时，我们希望使用多种卷积核，因为每个卷积核都可以从输入中得到一种特殊的模式或者概念，所以我们会有一组卷积核，我们通过第二个卷积核来得到第二个相同尺寸的绿色的激活映射。



如果我们这么做，比如我们有六个卷积核，然后我们对其做卷积操作，这样我们就得到了一个六层的激活映射，这六个映射相结合，可以得到 $28 \times 28 \times 6$ 尺寸的输出。

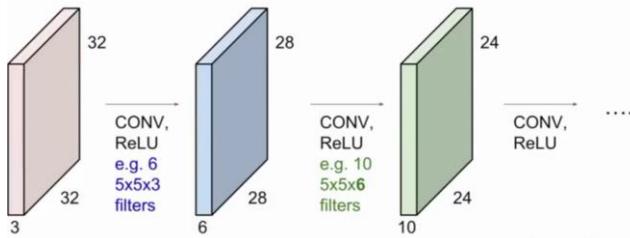
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



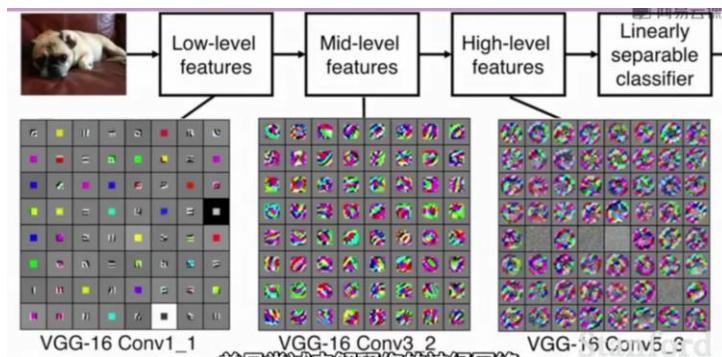
We stack these up to get a "new image" of size $28 \times 28 \times 6$!

我们来简述一下在卷积神经网络中我们是如何使用这些卷积层的，我们的 convnet 基本上是由多个卷积层组成的一个序列，它们依次堆叠，就像是神经网络中那样堆叠简单的线性层一样，之后我们将用激活函数对其进行逐一处理，比如一个 ReLU 的网络。我们会得到一个

CONV, ReLU 以及一些池化层，之后你会得到一系列的这些层，每一个都会有一个输出，而该输出又作为下一个卷积层的输入。

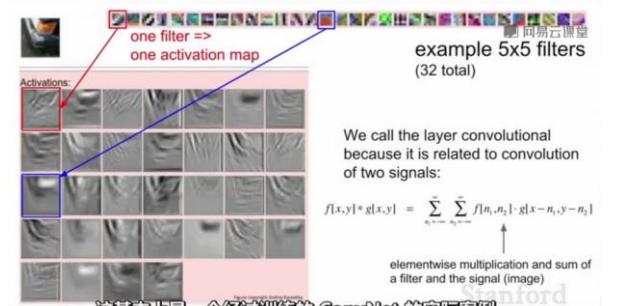


这些层使用了多个卷积核，每一个卷积核都会产生一个激活映射，我们可以观察这些层，然后将它们叠加成一个 ConvNet，最后的结果是你完成了对这组卷积核的学习，前面基层的卷积核一般代表了一些低阶的图像特征，比如说一些边缘特征等等，而中间层你可以得到一些更加复杂的图像特征，比如边角和斑点等等，而在更高阶的层，你会获得更丰富的内容。

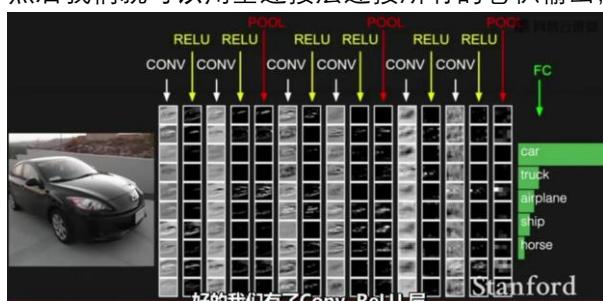


增加深度的原因是？

这主要是出于一种架构设计层面的考虑，但是初衷是，针对一个卷积神经网络，你可以选择不同的设计方式，你的卷积核尺寸大小，它滑动的步长，卷积核的数量等等。



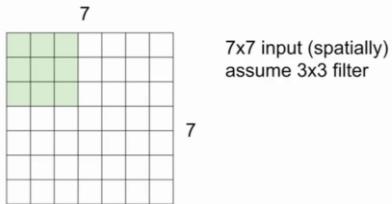
就是一个输入图片，让他通过很多层，第一个是卷积层，然后是非线性层，然后是池化层，这些措施已经大大降低了激活映射的采样尺寸，经过这些处理之后，最终得到卷积层输出，然后我们就可以用全连接层连接所有的卷积输出，并用其获得一个最终的分值函数，



具体是如何工作的？

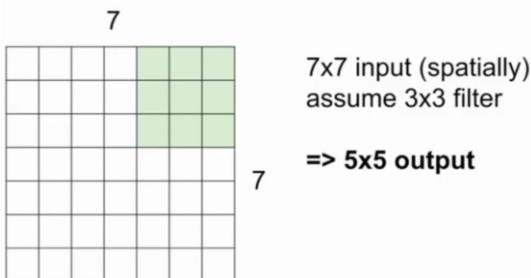
A closer look at spatial dimensions:

网易云课堂

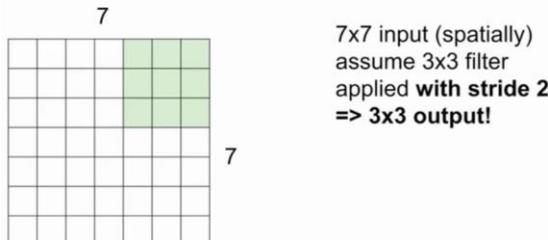


假如我们的输入是 7*7 的，而卷积核是 3*3 的，

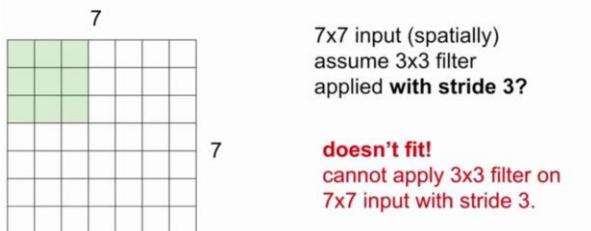
将其置于左上角，然后对相应的元素做点积，得到输出；然后我们向右移动一个像素，得到第二个值，迭代这样的操作，可以得到五个输出。



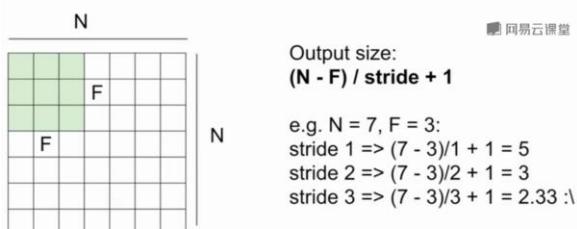
这时我们用 2 作为步长，我们先找到初始位置左上角，然后移动两个像素，得到输出。迭代操作得到激活为 3*3。



而使用 3 作为步长时，结果如何？结果是他在那里就不能拟合了，我们不会做这样的卷积操作，因为它会导致不对称的输出。



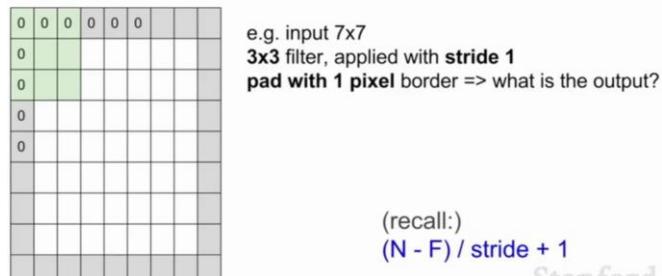
我们归纳一下公式：



用 0 来填充边界来得到我们想要的大小。我们需要在卷积核滑动初始位置的角上做什么呢？所以实际上发生的是我们在用零填充我们的输入图片，所以我们现在将能够在实际输入图像的右上方角位置放置一个卷积核。

在下面的例子中，我们会得到规格是多少的输出？是 7×7 的，我们先前的公式应用的前提是没有填充 0，在下面的例子中，这里的 N 不再是 7，而是 $1+7+1=9$ ，因此，应用公式时，我们要将填充的元素也算进去。这里的结果是 $7 \times 7 \times 1$ ，而深度则是我们采用的卷积核的数量，例如上面的例子，我们使用了 6 个卷积核，则最后的输出激活的深度就是 6

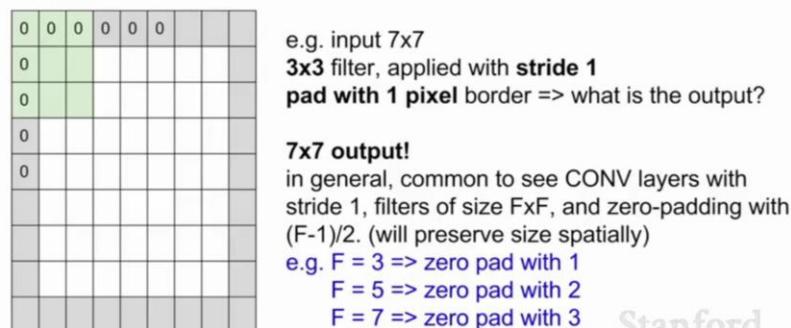
In practice: Common to zero pad the border



如果图片不是方形的，我们也可以采用纵横不同的步长，但这种方法不常用。

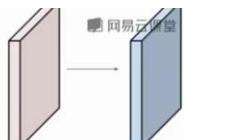
为何使用零填充？如果我们想要进行卷积操作后还能得到与输入同尺寸的输出，那我们就可以使用零填充，填充的 0 的数量取决于我们想要的输出尺寸，即目的是保持全尺寸输出。

In practice: Common to zero pad the border



使用卷积核会使得图片迅速的缩小，这会使得图片的边角信息会丢掉很多。

Examples time:



Input volume: **32x32x3**
10 5×5 filters with stride 1, pad 2

Output volume size: ?

激活映射 $32 \times 32 \times 10$

Output volume size:

$(32+2*2-5)/1+1 = 32$ spatially, so

32x32x10

Number of parameters in this layer? 参数有多少?

each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias)

=> **76 * 10 = 760**

我们有 5×5 的卷积核，乘上权重

$* 3(\text{输入深度}) + 1$ (偏差项参数数量)] * 10 (卷积核数量) = 760.

K 为卷积核数量，我们通常使用 2 的次幂个卷积核，然后是卷积核的尺寸 F ，步长 S ，0 填充的向量 P ，

Common settings:



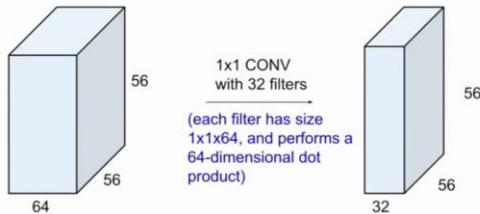
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d -th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias.



我们也可以使用 1×1 的卷积，虽然在看起来没有意义，但是我们控制卷积核的数量，就可以进行深度上的卷积，如下例采使用 1×1 的 32 个卷积核，就可以得到长宽不变但是深度为 32 的激活响应。

(btw, 1×1 convolution layers make perfect sense)

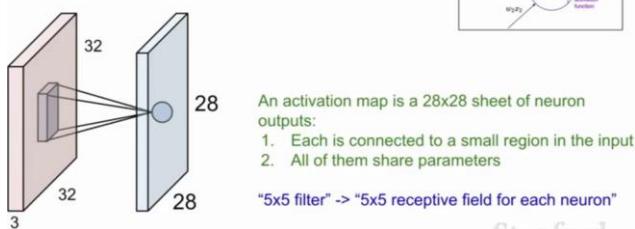
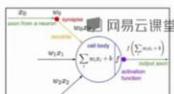


如何选用步长？

这跟图片的分辨率有关，当我们使用步长较大时，我们得到的是一个降采样之后的图片，这个降采样之后的图片，可以说是一种池化处理，不过又比池化在某些时候更好些。因为你能获得和降采样处理图片相同的效果，而且会降低在每一层中你所处理的激活映射的尺寸，并且还会影响到你所有用到的参数的数量，

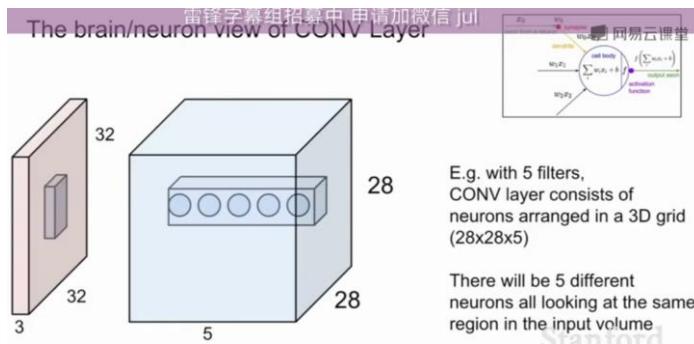
术语： 5×5 的卷积核，我们也可以称其为这个神经元的 5×5 的感受野，就是指输入区域，就是这个神经元所能接受到的视野。

The brain/neuron view of CONV Layer

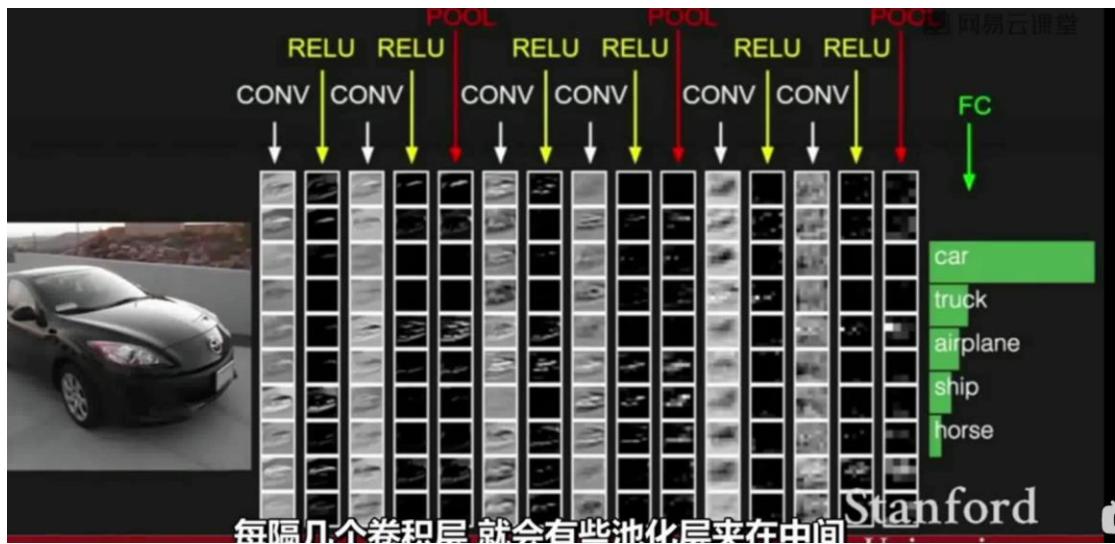


卷积核的数量就是所谓的深度，比如说，一共有五层的卷积核，那经过卷积操作后得到的激活响应的深度就是 5。我们来看一下这些卷积核，在激活量的某个空间区域内，沿着深度方

向，这五个神经元



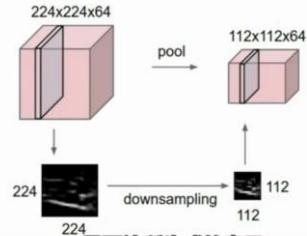
我们有卷积层，每隔几个卷积层，中间又有池化层



那这些池化层所要做的就是要让所生成的表示更小更容易控制，从而使得处理的参数变小，这关系到最后我们得到的参数数量，而且基本上也会关系到给定区域的不变性问题。但是池化层仅仅是在做降采样处理吗？我们只会做平面上的池化处理，而不做深度上的处理，因此

Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



输入的深度和输出的深度是一样的。

是要让所生成的表示

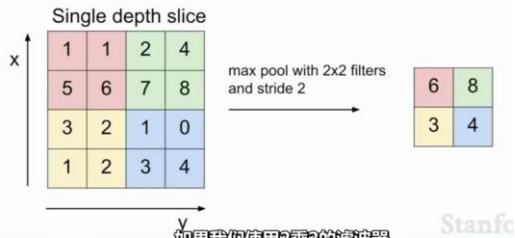
举一例，最常见的方法是最大池化法，在下面的例子中，池化层也有一个卷积核的大小，而且这个卷积核的大小和我们所要池化处理的区域大小是相同的，如果我们使用 2×2 的滤波器，这里我们设定步长为 2，我们滑动过滤器，但这次的过滤器不做点积操作，而是提取过滤部分的最大值，经过这样的操作过后，我们得到右面的结果。

这样的设定步长的方法在池化层是很常用的，使得卷积核不会互相重叠，我们希望进行的是降采样的操作，因此我们给定一个区域这样避免重叠的处理，然后只用一个值来表示整个区域是有道理的，然后我们就走进下一个区域，得到结果。

为什么是最大值池化？而非均值池化之类的操作？我们可将最大值池化看作是卷积核在在

某个区域的激活程度，如果我们要做检测之类的操作，用最大值池化来激活他是个很常用的方法。

既然池化和步长滑动都是降采样操作，能否只滑动步长而不池化，抑或是相反操作呢？实际上在实践中这样的想法是可行的。



我们得到下面尺寸的输入，然后设置超参数来选择卷积核的尺寸（或池化的范围）和滑动步长，然后我们计算输出的容量，公式与前面的卷积操作一样。需要注意的是，一般不在池化层做零填充，因为我们只做降采样，这样就不会导致卷积核扫过边缘，有一部分超出了输入的范围，池化层的典型设置是 2×2 的卷积核加步长 2 或者 3×3 加步长 3

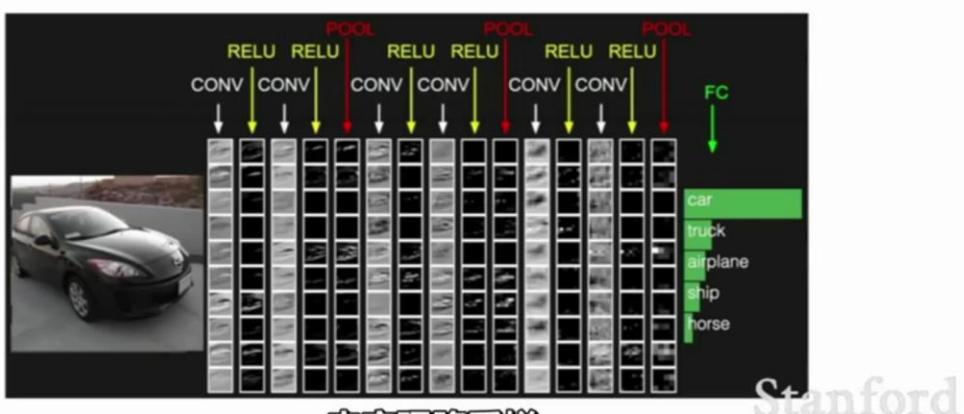
Common settings:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
 - their spatial extent F ,
 - the stride S ,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

$$\begin{aligned} F &= 2, S = 2 \\ F &= 3, S = 2 \end{aligned}$$

解释一下整个的工作过程，我们有一张汽车的图片，然后我们利用卷积核滑动整幅图像，得到其卷积后的激活映射，然后我们把激活映射送到 ReLU，得到一些这样的值，然后再进行卷积操作，ReLU 操作（对特征进行评分，与 0 做比较），然后我们对 ReLU 得到的结果进行池化降采样操作，取出每个卷积核对应的像素区域的最大值，然后我们就对其进行朴素的神经网络评分----对最后的矩阵做拉伸操作得到一列向量，得到全连接层，然后用 W 来处理这个向量，即最后的向量，即每个分类的得分。

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



似乎我们只获得了很小的信息量，那我们是如何进行评分的？

每一个池化层输出的值，实际上，是数据经过了整个网络处理后累积的结果，在最顶层，每个值都表示了上一阶的某个概念，比如上图中的第一列卷积操作，我们得到一般是某些特定的激活表征，比如边缘在图像各个位置的体现等等，越往后，我们得到就越加复杂，是更高层的表示，比如图片中的一些边角区域，所有的这些都有所含义，输入不再是原始图片，而是之前的输出，比如边缘的表示图作为下面操作的输入，它在边缘图上进一步的计算，从而得到更复杂的内容，检测到更复杂的物体，持续做这样的操作，这样当你到达最后的一个池化层的时候，每个值都代表了一组复合模板的激活情况，这样得到的全连接层会将所有的信息聚合到一起，得到一组分类的分值，每个值代表了复合的复杂概念的受激程度。

Summary

网易云课堂

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Typical architectures look like
 $[(CONV-RELU)^*N-POOL?]^*M-(FC-RELU)^*K,SOFTMAX$
where N is usually up to ~5, M is large, $0 \leq K \leq 2$.
 - but recent advances such as ResNet/GoogLeNet challenge this paradigm

训练神经网络—激活函数

最小批量随机梯度下降，我们通过使用计算图或者神经网络将数据进行正向传播，然后得到损失，通过整个网络的反向传播来计算梯度，使用这个梯度来更新网络中的参数或权重

Where we are now...

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

可供选择的非线性函数

Activation Functions

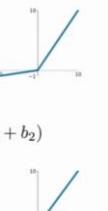
Sigmoid

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



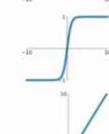
Leaky ReLU

$$\max(0.1x, x)$$



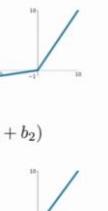
tanh

$$\tanh(x)$$



Maxout

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



ReLU

$$\max(0, x)$$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

Stanford

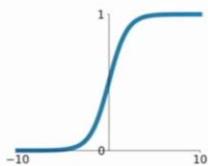
Overview

1. **One time setup**
activation functions, preprocessing, weight initialization, regularization, gradient checking
2. **Training dynamics**
babysitting the learning process, parameter updates, hyperparameter optimization
3. **Evaluation**
model ensembles

将每个元素输入到 sigmoid 非线性函数中，每个元素被压缩到[0,1]范围内，得到的值和输入成正比，sigmoid 函数可以看作一种神经元的饱和放电率，

Activation Functions

$$\sigma(x) = 1/(1 + e^{-x})$$



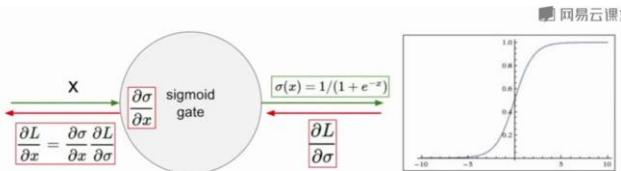
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

Sigmoid

3 problems:

饱和神经元将使得梯度消失。

1. Saturated neurons "kill" the gradients



What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

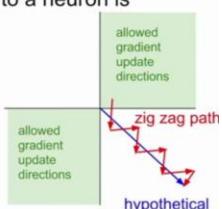
当 x 取极大的正值或负值时，会使得梯度消失，无法得到梯度流的反馈。

2. Sigmoid outputs are not zero-centered

sigmoid 是一个非零中心的函数。

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on w ?

Always all positive or all negative : (

(this is also why you want zero-mean data!)

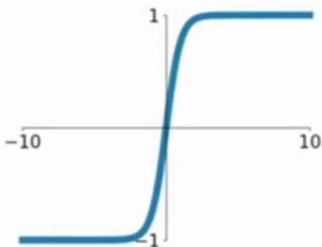
Stanford

如果我们的输入全都是正数，则得到的关于 w 的梯度都是正数或者全都是负数，也就是说我们使用链式原则时，将上游的符号传递了下来。来看上面的右图，如果过我们输入全都是正数，则梯度可以存在的象限只有一三象限，这是由梯度更新来得到的两个方向。在这种情况下，我们假设最佳的 w 实际上视图中的蓝色向量，我们从已知的一个点或者从红色箭头的起始端的顶部，我们不能沿着 w 这个方向直接求梯度，因为这个方向不是允许的两个梯度方向中的一个，因此我们只能沿着允许的两个方向来更新梯度，比如图中的这些红色箭头的方向。这也是为什么我们使用均值为 0 的数据。这样我们就可以得到正和负的数值，这样可以避免这种问题。

3. $\exp()$ is a bit compute expensive

使用指数函数，计算代价较大。但在神经网路种卷积核点乘的计算代价会更大，这种计算代价问题不大。

Activation Functions

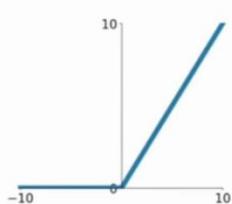


- Squashes numbers to range [-1, 1]
- zero centered (nice)
- still kills gradients when saturated :(

tanh(x)

Tanh 函数以 0 为中心，使得 sigmoid 的第二个问题不复存在，但是在其饱和时仍然会有梯度消失的问题。在这些区域内我们仍然会看到梯度基本上是扁平的，因此这将会阻止梯度的传递。因此即使是改进了 sigmoid 函数，但是仍然存在部分问题。

Activation Functions



- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

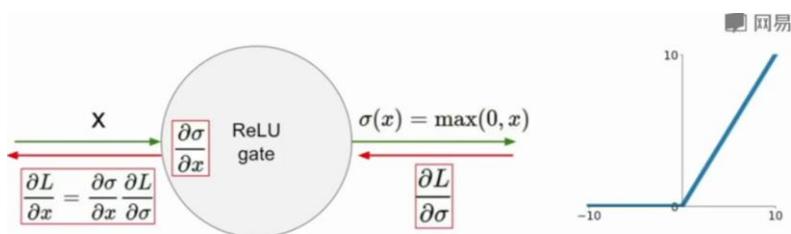
ReLU

(Rectified Linear Unit)

我们可以观察到 ReLU 在正的区域不会饱和，因此不会出现梯度消失的情况。而且其计算成本相较于其他函数更低，在实际操作中，我们一般会使用 ReLU，因为他比 Sigmoid 和 tanh 收敛的快得多，大约是六倍。

- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$? ReLU 不是以 0 为中心的了，而且 ReLU 在负半轴上的情况完全不同

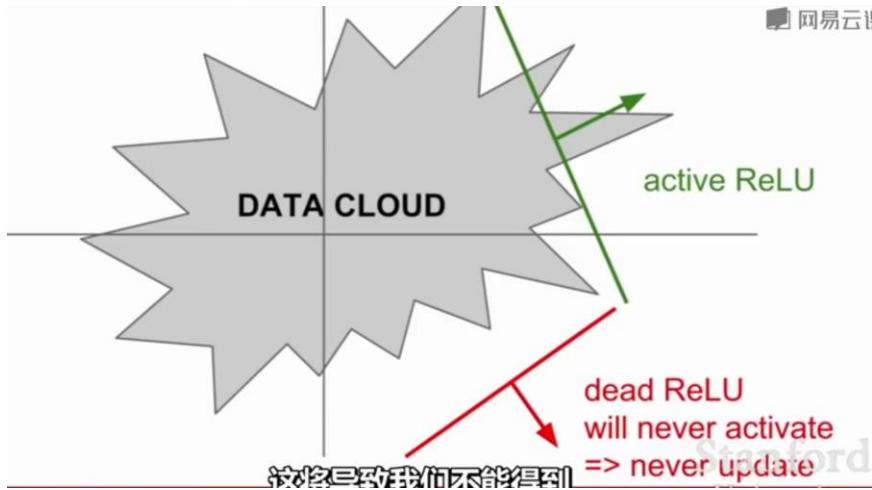


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

如果我们数据初始化和设置权重一开始做的不好，那就可能出现 dead ReLU 的情况，使得数据云中的一部分数据不会被激活。



通过观察数据云，我们如何得知 ReLU 是否会挂掉。我们使用很简单的 $x_1W_1+x_2w_2$ ，这里定义了分割的超平面，然后我们从中取出一半，这些为正，而另一半则会因为 ReLU 被抛弃。在实际中，我们偏向于使用较小的正偏置来初始化 ReLU，以增加它在初始化时被激活的可能性，并且获得一些更新，这基本上只是让更多的 ReLUs 在一开始就能放电的偏置项。

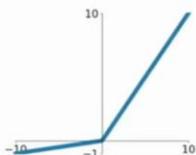
=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

大多数时候我们只是将偏置项初始化为 0。

Activation Functions

[Mass et al., 2013]
[He et al., 2015]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

我们对 ReLU 做出改进，

使得负半轴也有斜率。这时候，我们的函数不会饱和，效率很高，不会挂掉，Leaky ReLU 基本解决了 ReLU 有的问题。



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

Stanford

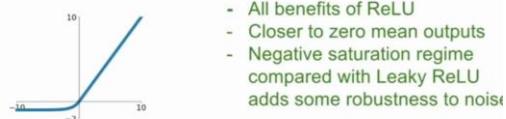
这里我们使用了

参数整流器，简称 PReLU，与 Leaky ReLU 类型，但是负半轴的参数是由 α 参数来确定的，因此我们不需要指定他，不需要硬编码他，而是把它当作一个可以反向传播和学习的参数，这给了他更多的灵活性。

指数线性单元，简称 ELU Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$ 这个函数有 ReLU 的优点，而且其输出均值还接近于 0，ELU 没有在负半区倾斜，这使得负半区饱和，有人认为，这会使得其鲁棒性更强，得到质量更好的激活响应。但是引入了指数函数使得计算代价变大。

最大神经元，它的作用是泛化 ReLU 和 Leaky ReLU，因为你只是提取这两个线性函数的最大值，他给我们的就是另一个线性机制的操作，这种方式不会饱和也不会挂掉。问题在于，你会把每个神经元的参数数量翻倍，所以说如果每个神经元原来有权重集 W ，而现在有了 W_1 和 W_2 ，相当于原来的两倍。

Maxout “Neuron”

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :

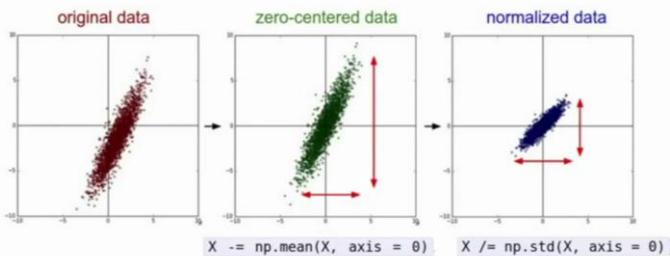
一般我们都使用 ReLU，我们需要非常谨慎地调整学习速率。尝试使用 ReLU 的变种，当然也可以使用 tanh，但是不要对他期盼太多，这些方法的实用性都较弱。记住不要使用 sigmoid

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

数据处理

我们得到数据后有可能会对其作零均值化数据，还有可能想要归一化数据（通过标准差），

Step 1: Preprocess the data



(Assume $X [NxD]$ is data matrix,
each example in a row)

Stanford

为什么要处理数据呢？在前面提到的非零中心化的问题，一般来说，即使不是全零或者全负，任何的偏差也有可能导致这个问题。

归一化数据，使得所有特征都在相同的值域内，并且这些特征贡献相同，我们在处理实际问题中，的确会做零中心化的处理，但我们不会真的去过多的归一化像素值，因为对于一般图像来说，就在这个位置你已经得到了相对可比较的范围与分布。我们坚持零均质化，而不做归一化，我们也不做更复杂的预处理。因为对于图像来说，我们并不是想要所有特征。我们在测试阶段也会做数据的预处理。在训练阶段我们会确定均值，并且把这个均值应用到测试数据中去，我们会用从训练数据中得到的相同经验均值来归一化。总的来说，一般对于图像，我们就是做零均值化的预处理。

TLDR: In practice for Images: center only

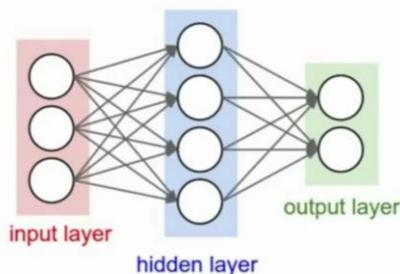
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
(mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
(mean along each channel = 3 numbers)

Not common to normalize
variance, to do PCA or
whitening Stanford

初始化网络权重

- Q: what happens when $W=0$ init is used?



如果我们将 W 初始值设为 0，则所有的神

经元都会做一样的事情，他们会得到相同输出和相同的梯度，他们以相同的方式更新，我们会得到完全相同的神经元。而我们所期望得是不同的神经元学习到不同的知识。也就是说，当我们用相同的参数去初始化时，基本没有打破参数对称得问题。

梯度会针对不同的神经元得到不同的代价函数，依赖于他所连接得类。但如果你通过网络去观察所有的神经元，遍历整个网络，你基本上有很多神经元，用一样的方式连接在一起，有相同的更新。

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(D,H)
```

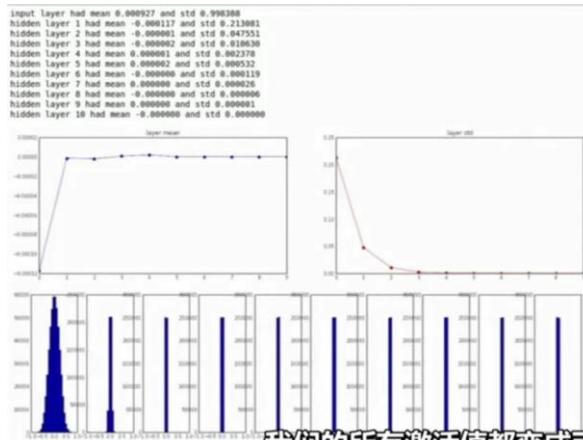
我们的第一个想法是将 W 权重置为一

个小的随机数，我们可以从一个概率分布中去抽样，这里采用了高斯分布，我们将其标准化后得到权重。

Works ~okay for small networks, but problems with
deeper networks.

这个矩阵在小型网络中还好，但

是在深度更高的网络中存在问题。



网易云课堂
All activations become zero!

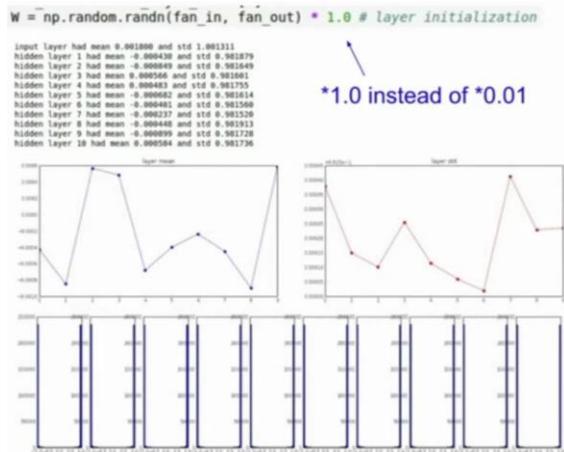
Q: think about the backward pass.
What do the gradients look like?

Hint: think about backward pass for a $W^T X$ gate.

Stanford

所有的激活值都变

成了 0。考虑反向传播，梯度会是什么样子的？我们每层都有很小的输入值。

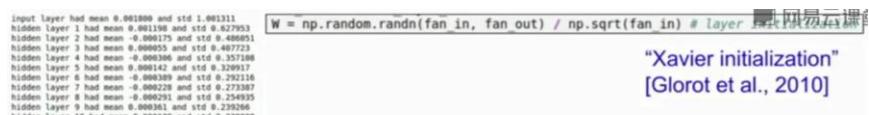


网易云课堂
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Stanford

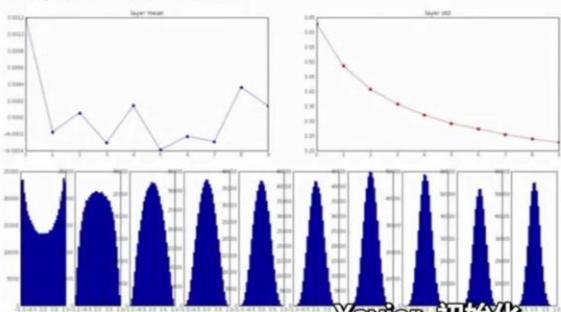
我们将 0.01 换成 1,

权重变大，使得 \tanh 的负方向或者正方向都可能饱和，使得激活趋向于 1 或者 -1 达到饱和，梯度将变成 0。由此可见，初始胡权重是一个复杂的事情，权重太小，网络崩溃，权重太大，网络饱和。经验上可行的方法是 Xavier 初始化。如果我们想要得到下面的输出样式，我们就需要通过高斯分布缩放来得到权重，通常我们要求输入的方差等于输出的方差，如果你推导出权重应该是多少，你就会得到这个公式，直觉上说，如果你有少量的输入，那么我们将除以较小的数，从而得到较大的权重。我们需要较大的权重，因为假设有少量的输入，用每一个输入值乘以权重，那么你需要一个更大的权重，来得到一个相同的输出方差。反之亦然，如果我们有很多的输入，我们想要更小的权重，以便让他在输出中得到相同的传播



网易云课堂
"Xavier initialization"
[Glorot et al., 2010]

Reasonable initialization.
(Mathematical derivation assumes linear activations)

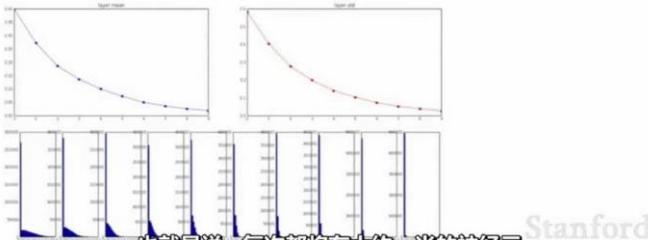


Stanford

如果我们使用 ReLU，每次都将由大约一半的神经元被设置为 0，它实际上是把你得到的方差减半。

`Input Layer had mean 0.000001 and std 0.000004
hidden Layer 1 had mean 0.000022 and std 0.000277
hidden Layer 2 had mean 0.172252 and std 0.463795
hidden Layer 3 had mean 0.334442 and std 0.832252
hidden Layer 4 had mean 0.136442 and std 0.398655
hidden Layer 5 had mean 0.472254 and std 0.187280
hidden Layer 6 had mean 0.449775 and std 0.877240
hidden Layer 7 had mean 0.449775 and std 0.877240
hidden Layer 8 had mean 0.425494 and std 0.839582
hidden Layer 9 had mean 0.312448 and std 0.839676
hidden Layer 10 had mean 0.312448 and std 0.839676`

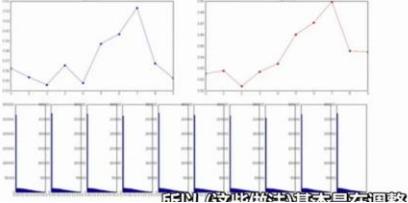
but when using the ReLU
nonlinearity it breaks.



Stanford

`Input Layer had mean 0.000001 and std 0.000004
hidden Layer 1 had mean 0.000022 and std 0.000277
hidden Layer 2 had mean 0.172252 and std 0.463795
hidden Layer 3 had mean 0.334442 and std 0.832252
hidden Layer 4 had mean 0.136442 and std 0.398655
hidden Layer 5 had mean 0.472254 and std 0.187280
hidden Layer 6 had mean 0.449775 and std 0.877240
hidden Layer 7 had mean 0.449775 and std 0.877240
hidden Layer 8 had mean 0.425494 and std 0.839582
hidden Layer 9 had mean 0.312448 and std 0.839676
hidden Layer 10 had mean 0.312448 and std 0.839676`

He et al., 2015
(note additional /2)



Stanford 可以通过/2 来解决这个问题。这些做法基本上

上是在调整，因为有一半的神经元被置 0，（和之前未用 ReLU 激活函数相比）等效的输入，实际上只有一半的输入，所以除以 2 就能很好的解决这个问题。

批量归一化

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

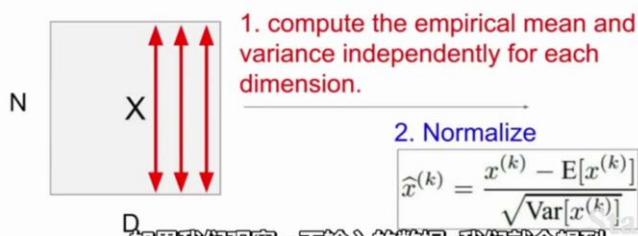
$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function.

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."

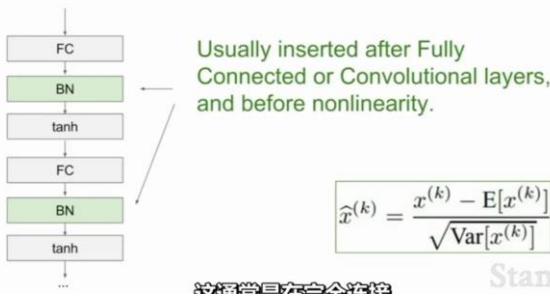


假设我们在当前的批处理中有 N 个训

练习样本，并且假设每批是 D 维的，我们将对每个维度独立计算经验均值和方差，所以基本上每个特征元素，通过批量处理我们都进行计算过了，我们的小批处理并且对其进行归一化。

Batch Normalization

[Ioffe and Szegedy, 2015]



$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

Stanford

这通常是在完全连接或卷积层之后插入的。

我们不断地在这些层上乘以 W，虽然会对每一层造成不好的尺度效应，但是这基本上可以消除这种影响，因为我们基本上是通过每个与神经元，激活函数相连的输入来进行缩放，所以我们可以用相同的方法来完全连接卷积层，唯一的区别是在卷积层的情况下，我们不仅想要归一化所有的特征维度的独立训练实例，而且在我们的激活映射图，包括我们的训练实例，我们也想要归一化跨特征维度和空间位置的所有特征。并且我们这样做的原因是因为我们想要服从卷积的性质，我们希望附近的位置能够以同样的方式进行归一化，所以在卷积层的情况下，在每个激活图中，我们会有一个均值和一个标准差，我们将在批处理的所有实例中进行归一化。

我们并不清楚要在每个全连接层之后进行批量归一化操作，也不清楚我们是否确实想要给这些 tanh 非线性函数输入一单位高斯数据，因为这里所做的是把输入限制，以避免出现饱和的情况。

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$$

$$\beta^{(k)} = E[x^{(k)}]$$

to recover the identity mapping.

所以我们强调的是批量归一化就是在完

成归一化操作之后，需要进行额外的缩放操作，所以我们先做了归一化，然后使用常量γ来进行缩放，在使用另外一个因子β进行平移，并且这里实际在做的是这样允许你回复恒等函数。如果网络需要，它可以学习缩放因子γ，使之等于方差，也可以学习β使之等于均值，在这种情况下你就可以恢复恒等映射，就像你没有进行批量归一化一样，所以现在你拥有让网络为了达到比较好的训练效果，去学习控制让 tanh 具有更高或耕地饱和度的能力。

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;
Parameters to be learned: γ, β

Output: $\{y_i = BN_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Stanford 我们总结一下批量归一化

的思想，我们提供输入，然后计算小批量均值，对每个输入的小批量都做这个操作，然后计算方差，通过均值和方差进行归一化，然后还有额外的缩放和平移因子，从而改进了整个网络的梯度流，还具有更高的鲁棒性，它能够在更广范围的学习率和不同初始值下工作。所以我们发现一旦使用了批量归一化，训练会变得更加容易。我们也可以把它当作正则化的一种方法。