Contents

[Contents 1](#_Toc488156844)

[Deep learning framework 1](#_Toc488156845)

[Keras 2](#_Toc488156846)

[安装 3](#_Toc488156847)

[概念 5](#_Toc488156848)

[workflow 22](#_Toc488156849)

[example 23](#_Toc488156850)

[Model 23](#_Toc488156851)

[tensorflow 24](#_Toc488156852)

[Machine learning 25](#_Toc488156853)

[MLP (multi-layer perception) 25](#_Toc488156854)

[deep learning 26](#_Toc488156855)

[Introduction to Deep Learning Algorithms 26](#_Toc488156856)

[一天搞懂深度学习 –李宏毅 27](#_Toc488156857)

[NN 27](#_Toc488156858)

[CNN 28](#_Toc488156859)

[RNN (Recurrent Neural Network) 29](#_Toc488156860)

[CS231n Convolutional Neural Networks for Visual Recognition 29](#_Toc488156861)

[NN (Neural Network) 29](#_Toc488156862)

[CNN (Convolutional Neural Networks / ConvNets) 33](#_Toc488156863)

[DeepLearning 37](#_Toc488156864)

# Deep learning framework

|  |  |  |  |
| --- | --- | --- | --- |
|  | By | interface | description |
| Caffe2 | the Berkeley Vision and Learning Center (BVLC) | C, C++, Python, MATLAB, cmd | deep learning framework  made with expression, speed, and modularity in mind |
| CNTK | The Microsoft Cognitive Toolkit | Python, C++, C# and cmd | unified deep-learning toolkit  train and combine popular model types across multiple GPUs and servers |
| TensorFlow | Google’s Machine Intelligence research organization | C++, Python | software library  numerical computation using data flow graphs |
| theano |  | Python | math expression compiler  defines, optimizes, and evaluates mathematical expressions |
| Torch |  | C, C++, Lua | scientific computing framework  offers wide support for machine learning algorithms. |
| Keras |  | Python | neural networks library  running on top of either TensorFlow or Theano, fast experimentation |

Monitor GPU resource

$ watch -n 1 nvidia-smi

## Keras

<https://keras-cn.readthedocs.io/en/latest/>

Keras的核心数据结构是“模型”,Keras的底层库使用Theano或TensorFlow(“符号主义”的库)

与传统的Python代码区别？

符号主义的计算首先定义各种变量，然后建立“计算图”，计算图规定了各个变量之间的计算关系。建立好的计算图需要编译已确定其内部细节，然而，此时的计算图还是一个“空壳子”，里面没有任何实际的数据，只有当你把需要运算的输入放进去后，才能在整个模型中形成数据流，从而形成输出值

深度学习的优化算法，说白了就是梯度下降。每次的参数更新有两种方式：

Batch gradient descent（批梯度下降）：遍历全部数据集算一次损失函数，然后算函数对各个参数的梯度，更新梯度。缺点：计算量开销大，计算速度慢，不支持在线学习

stochastic gradient descent（随机梯度下降）：速度快，但收敛性不好，可能在最优点附近晃来晃去，hit不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈

mini-batch gradient decent（小批的梯度下降）：数据分为若干批，按批来更新参数，批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性

张量可以看作是向量、矩阵的自然推广，我们用张量来表示广泛的数据类型

0阶张量，即标量，也就是一个数

1阶张量，也就是一个向量

2阶张量，也就是一个矩阵

3阶张量，一个立方体

张量的阶数有时候也称为维度，或者轴

Ubuntu 16.04 LTS是Nvidia官方以及绝大多数深度学习框架默认开发环境

Theano: a compiler for mathematical expressions in Python

TensorFlow: a python library for fast numerical computing

Keras: library addresses these concerns by providing a wrapper for both Theano and Tensorflow

scikit-learn library：general purpose machine learning framework in Python built on top

of SciPy

application checkpointing

dropout

convolutional neural networks

### 安装

# 系统升级

$ sudo apt update

$ sudo apt upgrade

# 安装python基础开发包

$ sudo apt install -y python-dev python-pip python-nose gcc g++ git gfortran vim

# 安装运算加速库

$ sudo apt install -y libopenblas-dev liblapack-dev libatlas-base-dev

# 安装CUDA开发环境

http://shomy.top/2016/12/29/gpu-tensorflow-install/

$ sudo dpkg -i cuda-repo-ubuntu1604-8-0-local\_8.0.44-1\_amd64.deb

$ sudo apt update

$ sudo apt install cuda

Copy cuDNN to /usr/local

sudo cp cuda/lib64/\* /usr/local/ cuda-8.0/lib64/

sudo cp cuda/include/cudnn.h /usr/local/ cuda-8.0/include/

# 将CUDA路径添加至环境变量

$ sudo gedit /etc/bash.bashrc

在bash.bashrc文件中添加：

export CUDA\_HOME=/usr/local/cuda-8.0

export PATH=/usr/local/cuda-8.0/bin${PATH:+:${PATH}}

export LD\_LIBRARY\_PATH=/usr/local/cuda-8.0/lib64${LD\_LIBRARY\_PATH:+:${LD\_LIBRARY\_PATH}}

$ source /etc/bash.bashrc

在.bashrc中添加如上相同内容

$ sudo gedit ~/.bashrc

$ nvcc -v 测试nVidia cuda版本号

/usr/local/cuda/samples$sudo make all –j8 测试是否成功，运行某个sample

Keras框架搭建

$ sudo pip install -U --pre pip setuptools wheel

$ sudo pip install -U --pre numpy scipy matplotlib scikit-learn scikit-image

$ sudo pip install -U --pre h5py pyyaml

$ sudo pip install -U --pre theano

$ sudo pip install -U --pre tensorflow-gpu

$ sudo pip install -U --pre keras

TensorFlow(当使用TensorFlow为后端时) or Theano(当使用Theano作为后端时), Keras默认使用TensorFlow作为后端来进行张量操作

keras$ sudo python setup.py install

or $ sudo pip install keras

$ python 验证

>>> import theano

>>> import keras

Keras环境设置

修改默认keras后端: gedit ~/.keras/keras.json

配置theano文件: gedit ~/.theanorc

[global]

openmp=False

device = gpu

floatX = float32

allow\_input\_downcast=True

[lib]

cnmem = 0.8

[blas]

ldflags= -lopenblas

[nvcc]

fastmath = True

验证keras是否安装成功

>>>import keras

加速测试

keras/examples/$ python mnist\_mlp.py

### 概念

#### Keras模型

model.summary()：打印出模型概况

config = model.get\_config():返回包含模型配置信息的Python字典。

model = Model.from\_config(config): 模型也可以从它的config信息中重构回去

model.get\_layer()：依据层名或下标获得层对象

model.get\_weights()：返回模型权重张量的列表，类型为numpy array

model.set\_weights()：从numpy array里将权重载入给模型，要求数组具有与model.get\_weights()相同的形状。

json\_string =model.to\_json：返回代表模型的JSON字符串，仅包含网络结构，不包含权值。

model = model\_from\_json(json\_string): 可以从JSON字符串中重构原模型

model.save\_weights(filepath)：将模型权重保存到指定路径，文件类型是HDF5（后缀是.h5）

model.load\_weights(filepath, by\_name=False)：从HDF5文件中加载权重到当前模型中, 默认情况下模型的结构将保持不变。如果想将权重载入不同的模型（有些层相同）中，则设置by\_name=True，只有名字匹配的层才会载入权重

Sequential模型

model.layers: 是添加到模型上的层的list

add(self, layer): 向模型中添加一个层, Layer对象

compile(self, optimizer, loss, metrics=None, sample\_weight\_mode=None)

fit(...)

evaluate(...)

predict\_classes(self, x, batch\_size=32, verbose=1)

predict\_proba(self, x, batch\_size=32, verbose=1)

train\_on\_batch(self, x, y, class\_weight=None, sample\_weight=None)

predict\_on\_batch(self, x)

fit\_generator(...)

evaluate\_generator(...)

predict\_generator(…)

函数式模型Model

广义的拥有输入和输出的模型，用Model来初始化一个函数式模型

from keras.models import Model

model = Model(inputs=[a1, a2], outputs=[b1, b3, b3])

model.layers：组成模型图的各个层

model.inputs：模型的输入张量列表

model.outputs：模型的输出张量列表

其余api同Sequential模型

from keras.layers import Input, Dense

from keras.models import Model

# This returns a tensor

inputs = Input(shape=(784,))

**层对象接受张量为参数，返回一个张量。**

# a layer instance is callable on a tensor, and returns a tensor

x = Dense(64, activation='relu')(inputs)

x = Dense(64, activation='relu')(x)

predictions = Dense(10, activation='softmax')(x)

**输入是张量，输出也是张量的一个框架就是一个模型，通过Model定义。**

# This creates a model that includes the Input layer and three Dense layers

model = Model(inputs=inputs, outputs=predictions)

model.compile(optimizer='rmsprop',loss='categorical\_crossentropy', metrics=['accuracy'])

model.fit(data, labels) # starts training

**重用已经训练好的模型**

把模型当作一个层一样，通过提供一个tensor来调用它。注意当你调用一个模型时，你不仅仅重用了它的结构，也重用了它的权重。

x = Input(shape=(784,))

# This works, and returns the 10-way softmax we defined above.

y = model(x)

**多输入和多输出模型**

main\_input = Input(shape=(100,), dtype='int32', name='main\_input')

...

auxiliary\_output = Dense(1, activation='sigmoid', name='aux\_output')(main\_medi)

auxiliary\_input = Input(shape=(5,), name='aux\_input')

x = keras.layers.concatenate([main\_medi, auxiliary\_input])

...

main\_output = Dense(1, activation='sigmoid', name='main\_output')(x)

model = Model(inputs=[main\_input, auxiliary\_input], outputs=[main\_output, auxiliary\_output])

model.compile(optimizer='rmsprop',loss={'main\_output': 'binary\_crossentropy', 'aux\_output': 'binary\_crossentropy'},

loss\_weights={'main\_output': 1., 'aux\_output': 0.2})

# And trained it via:

model.fit({'main\_input': headline\_data, 'aux\_input': additional\_data},

{'main\_output': labels, 'aux\_output': labels},

epochs=50, batch\_size=32)

**共享层**

把一个相同的Conv2D应用于一个大小为(3,32,32)的数据，然后又将其应用于一个(3,64,64)的数据，那么此时该层就具有了多个输入和输出的shape

a = Input(shape=(3, 32, 32))

b = Input(shape=(3, 64, 64))

conv = Conv2D(16, (3, 3), padding='same')

conved\_a = conv(a)

# Only one input so far, the following will work:

assert conv.input\_shape == (None, 3, 32, 32)

conved\_b = conv(b)

# now the `.input\_shape` property wouldn't work, but this does:

assert conv.get\_input\_shape\_at(0) == (None, 3, 32, 32)

assert conv.get\_input\_shape\_at(1) == (None, 3, 64, 64)

层Layer

layer = Dense(32)

reconstructed\_layer = Dense.from\_config(config)层也可以借由配置信息重构

layer.get\_weights()：返回层的权重（numpy array）

layer.set\_weights(weights)：从numpy array中将权重加载到该层中，要求numpy array的形状与\* layer.get\_weights()的形状相同

config = layer.get\_config()：返回当前层配置信息的字典

获取层输入输出信息

非共享层

layer.input 输入张量

layer.output 输出张量

layer.input\_shape 输入数据的形状

layer.output\_shape 输出数据的形状

层有多个计算节点

layer.get\_input\_at(node\_index)

layer.get\_output\_at(node\_index)

layer.get\_input\_shape\_at(node\_index)

layer.get\_output\_shape\_at(node\_index)

预训练权重的Keras模型

模型的预训练权重将下载到~/.keras/models/并在载入模型时自动载入,这些模型可以用来进行预测、特征提取和finetune

ResNet50模型: from keras.applications.resnet50 import ResNet50

VGG16模型: from keras.applications.vgg16 import VGG16

model = VGG16(weights='imagenet', include\_top=False)

img = image.load\_img(img\_path, target\_size=(224, 224))

x = image.img\_to\_array(img)

x = preprocess\_input(x)

features = model.predict(x)

VGG19模型: from keras.applications.vgg19 import VGG19

InceptionV3模型: from keras.applications.inception\_v3 import InceptionV3

Xception模型: from keras.applications.xception import Xception

模型可视化 (依赖 pydot-ng 和 graphviz)

from keras.utils import vis\_utils, plot\_model

plot\_model(model, to\_file='model.png')

在ipython中展示图片

from IPython.display import SVG

from keras.utils.visualize\_util import model\_to\_dot

SVG(model\_to\_dot(model).create(prog='dot', format='svg'))

#### 常用层

对应于core模块，core内部定义了一系列常用的网络层，包括全连接、激活层等

from keras.layers.core import Dense, Dropout, Flatten, Reshape, Permute, RepeatVector, Lambda, ActivityRegularizer, Masking

Dense(units, activation, use\_bias, kernel\_initializer, bias\_initializer, kernel\_regularizer, bias\_regularizer, activity\_regularizer, kernel\_constraint, bias\_constraint)

kernel\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。

bias\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器。

kernel\_regularizer：施加在权重上的正则项，为Regularizer对象

bias\_regularizer：施加在偏置向量上的正则项，为Regularizer对象

activity\_regularizer：施加在输出上的正则项，为Regularizer对象

kernel\_constraints：施加在权重上的约束项，为Constraints对象

bias\_constraints：施加在偏置上的约束项，为Constraints对象

Dropout(rate, noise\_shape=None, seed=None)

Dropout将在训练过程中每次更新参数时随机断开一定百分比（rate）的输入神经元，Dropout层用于防止过拟合。

Flatten()

Flatten层用来将输入“压平”，即把多维的输入一维化

Reshape(target\_shape)

Reshape层用来将输入shape转换为特定的shape

# as first layer in a Sequential model

model = Sequential()

model.add(Reshape((3, 4), input\_shape=(12,)))

# now: model.output\_shape == (None, 3, 4)

# note: `None` is the batch dimension

# as intermediate layer in a Sequential model

model.add(Reshape((6, 2)))

# now: model.output\_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension

model.add(Reshape((-1, 2, 2)))

# now: model.output\_shape == (None, 3, 2, 2)

Permute(dims)

Permute层将输入的维度按照给定模式进行重排，例如，当需要将RNN和CNN网络连接时，可能会用到该层。

model = Sequential()

model.add(Permute((2, 1), input\_shape=(10, 64)))

# now: model.output\_shape == (None, 64, 10)

# note: `None` is the batch dimension

RepeatVector(n)

RepeatVector层将输入重复n次

model = Sequential()

model.add(Dense(32, input\_dim=32))

# now: model.output\_shape == (None, 32)

# note: `None` is the batch dimension

model.add(RepeatVector(3))

# now: model.output\_shape == (None, 3, 32)

Lambda(function, output\_shape=None, mask=None, arguments=None)

本函数用以对上一层的输出施以任何Theano/TensorFlow表达式

def antirectifier(x):

pos = K.relu(x)

neg = K.relu(-x)

return K.concatenate([pos, neg], axis=1)

def antirectifier\_output\_shape(input\_shape):

shape = list(input\_shape)

shape[-1] \*= 2

return tuple(shape)

model.add(Lambda(antirectifier,

output\_shape=antirectifier\_output\_shape))

ActivityRegularization(l1=0.0, l2=0.0)

基于其激活值更新损失函数值,

l1：1范数正则因子（正浮点数）

l2：2范数正则因子（正浮点数）

#### 卷积层，反卷积，Cropping，UpSampling，ZeroPadding，Pooling

from keras.layers.convolutional import Conv1D, Conv2D, SeparableConv2D, Conv2DTranspose, Conv3D, Cropping1D, Cropping2D, Cropping3D, UpSampling1D, UpSampling2D, UpSampling3D, ZeroPadding1D, ZeroPadding2D, ZeroPadding3D

<http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html#transposed-convolution-arithmetic>

一维卷积层（即时域卷积），用以在一维输入信号上进行邻域滤波

Conv1D(...)

二维卷积层，即对图像的空域卷积。该层对二维输入进行滑动窗卷积

Conv2D(filters, kernel\_size, strides=(1, 1), padding='valid', data\_format=None, dilation\_rate=(1, 1), activation=None, use\_bias=True, kernel\_initializer='glorot\_uniform', bias\_initializer='zeros', kernel\_regularizer=None, bias\_regularizer=None, activity\_regularizer=None, kernel\_constraint=None, bias\_constraint=None)

filters：卷积核的数目（即输出的维度）

kernel\_size：卷积核的宽度和长度。如为单个整数，则表示在各个空间维度的相同长度。

strides：卷积的步长。如为单个整数，则表示在各个空间维度的相同步长。

padding：补0策略，为“valid”, “same” 。“valid”代表只进行有效的卷积，“same”代表保留边界处的卷积结果

activation：激活函数

dilation\_rate：指定dilated convolution中的膨胀比例。

data\_format：字符串，“channels\_first”或“channels\_last”之一，代表图像的通道维的位置。该参数是Keras 1.x中的image\_dim\_ordering，**“channels\_last”对应原本的“tf”，“channels\_first”对应原本的“th”。**

use\_bias:布尔值，是否使用偏置项

kernel\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器

bias\_initializer：权值初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的初始化器

kernel\_regularizer：施加在权重上的正则项，为Regularizer对象

bias\_regularizer：施加在偏置向量上的正则项，为Regularizer对象

activity\_regularizer：施加在输出上的正则项，为Regularizer对象

kernel\_constraints：施加在权重上的约束项，为Constraints对象

bias\_constraints：施加在偏置上的约束项，为Constraints对象

输入shape

**‘channels\_first’模式下，输入形如（samples, channels，rows，cols）的4D张量**

**‘channels\_last’模式下，输入形如（samples，rows，cols，channels）的4D张量**

输出shape

**‘channels\_first’模式下，为形如（samples，nb\_filter, new\_rows, new\_cols）的4D张量**

**‘channels\_last’模式下，为形如（samples，new\_rows, new\_cols，nb\_filter）的4D张量**

输出的行列数可能会因为填充方法而改变

可分离卷积: SeparableConv2D(...)

反卷积: Conv2DTranspose(...)

Conv3D(...)

三维卷积对三维的输入进行滑动窗卷积，例如input\_shape = (3,10,128,128)代表对10帧128\*128的彩色RGB图像进行卷积

Cropping1D(cropping=(1, 1))

在时间轴（axis1）上对1D输入（即时间序列）进行裁剪

Cropping2D(cropping=((0, 0), (0, 0)), data\_format=None)

对2D输入（图像）进行裁剪，将在空域维度，即宽和高的方向上裁剪

cropping：长为2的整数tuple，分别为宽和高方向上头部与尾部需要裁剪掉的元素数

Cropping3D(...)

UpSampling1D(size=2): 在时间轴上，将每个时间步重复length次

UpSampling2D(size=(2, 2), data\_format=None)

将数据的行和列分别重复size[0]和size[1]次

UpSampling3D(size=(2, 2, 2), data\_format=None)

ZeroPadding1D(padding=1):对1D输入的首尾端（如时域序列）填充0

ZeroPadding2D(padding=(1, 1), data\_format=None)

对2D输入（如图片）的边界填充0，以控制卷积以后特征图的大小

from keras.layers.pooling import MaxPooling1D, MaxPooling2D, MaxPooling3D, AveragePooling1D, AveragePooling2D, AveragePooling3D, GlobalMaxPooling1D, GlobalMaxPooling2D, GlobalAveragePooling2D

#### 局部连接层(locally-connected)，循环层(Recurrent),嵌入层(Embedding)

from keras.layers.local import LocallyConnected1D, LocallyConnected2D

LocallyConnected1D层与Conv1D工作方式类似，唯一的区别是不进行权值共享。即施加在不同输入位置的滤波器是不一样的。

# apply a 3x3 unshared weights convolution with 64 output filters on a 32x32 image

# with `data\_format="channels\_last"`:

model = Sequential()

model.add(LocallyConnected2D(64, (3, 3), input\_shape=(32, 32, 3)))

# now model.output\_shape == (None, 30, 30, 64)

# notice that this layer will consume (30\*30)\*(3\*3\*3\*64) + (30\*30)\*64 parameters

from keras.layers.recurrent import SimpleRNN, GRU, LSTM

SimpleRNN(...): 全连接RNN网络

GRU(...): 门限循环单元

LSTM(...): 长短期记忆模型

#### 融合层(Merge),激活层(Activation), 规范层(BatchNormalization), 噪声层(Noise),包装器(Wrapper)

from keras.layers import merge

Merge层提供了一系列用于融合两个层或两个张量的层对象和方法。以大写首字母开头的是Layer类，以小写字母开头的是张量的函数。小写字母开头的张量函数在内部实际上是调用了大写字母开头的层。

merge.Add(): 将Layer that adds a list of inputs.

merge.Multiply(): 逐元素积的张量

merge.Average(): 逐元素均值

merge.Maximum(): 逐元素最大值

merge.Concatenate(axis=-1): 按照给定轴相接构成的向量。

merge.Dot(axes, normalize=False):两张量乘积

例如，如果两个张量a和b的shape都为（batch\_size, n），则输出为形如（batch\_size,1）的张量，结果张量每个batch的数据都是a[i,:]和b[i,:]的矩阵（向量）点积。

normalize: 布尔值，是否沿执行成绩的轴做L2规范化，如果设为True，那么乘积的输出是两个样本的余弦相似性

from keras.layers import Activation

预定义激活函数

**softmax：对输入数据的最后一维进行softmax**

elu

softplus

softsign

relu

tanh

sigmoid

hard\_sigmoid

linear

可以通过传递一个逐元素运算的Theano/TensorFlow函数来作为激活函数

from keras import backend as K

def tanh(x):

return K.tanh(x)

model.add(Dense(64, activation=tanh))

model.add(Activation(tanh)

from keras.layers.advanced\_activations import LeakyReLU, PReLU, ELU, ThresholdedReLU

当不激活时，LeakyReLU仍然会有非零输出值，从而获得一个小梯度，避免ReLU可能出现的神经元“死亡”现象

from keras.layers.normalization import BatchNormalization

**该层在每个batch上将前一层的激活值重新规范化，即使得其输出数据的均值接近0，其标准差接近1**

（1）加速收敛

（2）控制过拟合，可以少用或不用Dropout和正则

（3）降低网络对初始化权重不敏感

（4）允许使用较大的学习率

from keras.layers.noise import GaussianNoise, GaussianDropout

GaussianNoise(stddev)

为数据施加0均值，标准差为stddev的加性高斯噪声。该层在克服过拟合时比较有用，你可以将它看作是**随机的数据提升**。高斯噪声是需要对输入数据进行破坏时的自然选择。

一个使用噪声层的典型案例是构建去噪自动编码器，即Denoising AutoEncoder（DAE）。该编码器试图从加噪的输入中重构无噪信号，以学习到原始信号的鲁棒性表示

GaussianDropout(rate)

为层的输入施加以1为均值，标准差为sqrt(rate/(1-rate)的乘性高斯噪声

from keras.layers.wrappers import TimeDistributed, Bidirectional

#### 自定义层

Keras2层的框架结构

from keras import backend as K

from keras.engine.topology import Layer

import numpy as np

#阅读源代码

class MyLayer(Layer):

def \_\_init\_\_(self, output\_dim, \*\*kwargs):

self.output\_dim = output\_dim

super(MyLayer, self).\_\_init\_\_(\*\*kwargs)

#定义权重的方法

def build(self, input\_shape):

# Create a trainable weight variable for this layer.

self.kernel = self.add\_weight(shape=(input\_shape[1], self.output\_dim),

initializer='uniform',

trainable=True)

super(MyLayer, self).build(input\_shape) # Be sure to call this somewhere!

#定义层功能的方法

def call(self, x):

return K.dot(x, self.kernel)

#指定shape变化的方法

def compute\_output\_shape(self, input\_shape):

return (input\_shape[0], self.output\_dim)

#### 数据预处理

from keras.preprocessing.image import ImageDataGenerator

ImageDataGenerator(featurewise\_center=False,

samplewise\_center=False,

featurewise\_std\_normalization=False,

samplewise\_std\_normalization=False,

zca\_whitening=False,

rotation\_range=0.,

width\_shift\_range=0.,

height\_shift\_range=0.,

shear\_range=0.,

zoom\_range=0.,

channel\_shift\_range=0.,

fill\_mode='nearest',

cval=0.,

horizontal\_flip=False,

vertical\_flip=False,

rescale=None,

preprocessing\_function=None,

data\_format=K.image\_data\_format())

用以生成一个batch的图像数据，支持实时数据提升。训练时该函数会无限生成数据，直到达到规定的epoch次数为止。

featurewise\_center：布尔值，使输入数据集去中心化（均值为0）, 按feature执行

featurewise\_std\_normalization：布尔值，将输入除以数据集的标准差以完成标准化, 按feature执行

rotation\_range：整数，数据提升时图片随机转动的角度

width\_shift\_range：浮点数，图片宽度的某个比例，数据提升时图片水平偏移的幅度

height\_shift\_range：浮点数，图片高度的某个比例，数据提升时图片竖直偏移的幅度

shear\_range：浮点数，剪切强度（逆时针方向的剪切变换角度）

zoom\_range：浮点数或形如[lower,upper]的列表，随机缩放的幅度，若为浮点数，则相当于[lower,upper] = [1 - zoom\_range, 1+zoom\_range]

fit(x, augment=False, rounds=1)：计算依赖于数据的变换所需要的统计信息(均值方差等),只有使用featurewise\_center，featurewise\_std\_normalization或zca\_whitening时需要此函数。

flow(self, X, y, batch\_size=32, shuffle=True, seed=None, ...)：接收numpy数组和标签为参数,生成经过数据提升或标准化后的batch数据,并在一个无限循环中不断的返回batch数据

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

y\_train = np\_utils.to\_categorical(y\_train, num\_classes)

y\_test = np\_utils.to\_categorical(y\_test, num\_classes)

datagen = ImageDataGenerator(

featurewise\_center=True,

featurewise\_std\_normalization=True,

rotation\_range=20,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

horizontal\_flip=True)

# compute quantities required for featurewise normalization

# (std, mean, and principal components if ZCA whitening is applied)

datagen.fit(x\_train)

# fits the model on batches with real-time data augmentation:

model.fit\_generator(datagen.flow(x\_train, y\_train, batch\_size=32),

steps\_per\_epoch=len(x\_train), epochs=epochs)

同时变换图像和mask

# we create two instances with the same arguments

data\_gen\_args = dict(featurewise\_center=True,

featurewise\_std\_normalization=True,

rotation\_range=90.,

width\_shift\_range=0.1,

height\_shift\_range=0.1,

zoom\_range=0.2)

image\_datagen = ImageDataGenerator(\*\*data\_gen\_args)

mask\_datagen = ImageDataGenerator(\*\*data\_gen\_args)

# Provide the same seed and keyword arguments to the fit and flow methods

seed = 1

image\_datagen.fit(images, augment=True, seed=seed)

mask\_datagen.fit(masks, augment=True, seed=seed)

image\_generator = image\_datagen.flow\_from\_directory(

'data/images',

class\_mode=None,

seed=seed)

mask\_generator = mask\_datagen.flow\_from\_directory(

'data/masks',

class\_mode=None,

seed=seed)

# combine generators into one which yields image and masks

train\_generator = zip(image\_generator, mask\_generator)

model.fit\_generator(train\_generator, steps\_per\_epoch=2000, epochs=50)

#### 性能评估

from keras import losses

真实的优化目标函数是在各个数据点得到的损失函数值之和的均值

可用的目标函数:

mean\_squared\_error或mse

mean\_absolute\_error或mae

mean\_absolute\_percentage\_error或mape

mean\_squared\_logarithmic\_error或msle

squared\_hinge

hinge

binary\_crossentropy（亦称作对数损失，logloss）

logcosh

categorical\_crossentropy：亦称作多类的对数损失，注意使用该目标函数时，需要将标签转化为形如(nb\_samples, nb\_classes)的二值序列

sparse\_categorical\_crossentrop：如上，但接受稀疏标签

kullback\_leibler\_divergence:从预测值概率分布Q到真值概率分布P的信息增益,用以度量两个分布的差异.

poisson：即(predictions - targets \* log(predictions))的均值

cosine\_proximity：即预测值与真实标签的余弦距离平均值的相反数

from keras import optimizers

optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

随机梯度下降法，支持动量参数，支持学习衰减率，支持Nesterov动量

optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-06)

除学习率可调整外，建议保持优化器的其他默认参数不变,递归神经网络时的一个良好选择

optimizers.Adagrad(lr=0.01, epsilon=1e-06): 建议保持优化器的默认参数不变

Adadelta(lr=1.0, rho=0.95, epsilon=1e-06): 建议保持优化器的默认参数不变

optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08):该优化器的默认值来源于参考文献

Nadam(lr=0.002, beta\_1=0.9, beta\_2=0.999, epsilon=1e-08, schedule\_decay=0.004)

Adam本质上像是带有动量项的RMSprop，Nadam就是带有Nesterov 动量的Adam RMSprop

from keras import metrics

metrics=[metrics.mae, metrics.categorical\_accuracy]

categorical\_accuracy:对多分类问题,计算所有预测值上的平均正确率

定制评估函数

#(y\_true, y\_pred) as arguments and return a single tensor value.

import keras.backend as K

def mean\_pred(y\_true, y\_pred):

return K.mean(y\_pred)

model.compile(optimizer='rmsprop',

loss='binary\_crossentropy',

metrics=['accuracy', mean\_pred])

不同的层可能使用不同的关键字来传递初始化方法

Dense(64, kernel\_initializer='random\_uniform', bias\_initializer='zeros')

from keras import initializers

kernel\_initializer=initializers.random\_normal(stddev=0.01)

kernel\_initializer='random\_normal'

预定义初始化方法

initializers.Zeros()

initializers.Ones()

Constant(value=0)

RandomNormal(mean=0.0, stddev=0.05, seed=None))

RandomUniform(minval=-0.05, maxval=0.05, seed=None)

TruncatedNormal(mean=0.0, stddev=0.05, seed=None):截尾高斯分布初始化，该初始化方法与RandomNormal类似，但位于均值两个标准差以外的数据将会被丢弃并重新生成，形成截尾分布。该分布是神经网络权重和滤波器的推荐初始化方法。

VarianceScaling(scale=1.0, mode='fan\_in', distribution='normal', seed=None):自适应目标张量的shape。

Orthogonal(gain=1.0, seed=None):用随机正交矩阵初始化

Identity(gain=1.0)

lecun\_uniform(seed=None): LeCun均匀分布初始化方法

glorot\_normal(seed=None):Glorot正态分布初始化方法，也称作Xavier正态分布初始化

he\_normal(seed=None):He正态分布初始化方法，也称作Xavier正态分布初始化

he\_uniform(seed=None)

自定义初始化器

from keras import backend as K

def my\_init(shape, dtype=None):

return K.random\_normal(shape, dtype=dtype)

model.add(Dense(64, init=my\_init))

#### 正则项，约束项

正则项在优化过程中层的参数或层的激活值添加惩罚项，这些惩罚项将与损失函数一起作为网络的最终优化目标

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但Dense, Conv1D, Conv2D, Conv3D具有共同的接口。

kernel\_regularizer：施加在权重上的正则项，为keras.regularizer.Regularizer对象

bias\_regularizer：施加在偏置向量上的正则项，为keras.regularizer.Regularizer对象

activity\_regularizer：施加在输出上的正则项，为keras.regularizer.Regularizer对象

from keras import regularizers

model.add(Dense(64, input\_dim=64,

kernel\_regularizer=regularizers.l2(0.01),

activity\_regularizer=regularizers.l1(0.01)))

开发新的正则项

任何以权重矩阵作为输入并返回单个数值的函数均可以作为正则项

from keras import backend as K

def l1\_reg(weight\_matrix):

return 0.01 \* K.sum(K.abs(weight\_matrix))

Dense(64, input\_dim=64, kernel\_regularizer=l1\_reg)

约束项

在优化过程中为网络的参数施加约束

惩罚项基于层进行惩罚，目前惩罚项的接口与层有关，但Dense, Conv1D, Conv2D, Conv3D具有共同的接口。

kernel\_constraint：对主权重矩阵进行约束

bias\_constraint：对偏置向量进行约束

from keras.constraints import maxnorm

Dense(64, kernel\_constraint=max\_norm(2.))

预定义约束项

max\_norm(m=2)：最大模约束

non\_neg()：非负性约束

unit\_norm()：单位范数约束, 强制矩阵沿最后一个轴拥有单位范数

#### 回调函数Callbacks

回调函数是一组在训练的特定阶段被调用的函数集，你可以使用回调函数来观察训练过程中网络内部的状态和统计信息。

回调函数以字典logs为参数，该字典包含了一系列与当前batch或epoch相关的信息。

在每个epoch的结尾处（on\_epoch\_end），logs将包含训练的正确率和误差，acc和loss，如果指定了验证集，还会包含验证集正确率和误差val\_acc)和val\_loss，val\_acc还额外需要在.compile中启用metrics=['accuracy']。

在每个batch的开始处（on\_batch\_begin）：logs包含size，即当前batch的样本数

在每个batch的结尾处（on\_batch\_end）：logs包含loss，若启用accuracy则还包含acc

from keras import callbacks

callbacks.BaseLogger(): 对每个epoch累加metrics指定的监视指标的epoch平均值, 在Keras模型中会被自动调用

callbacks.History(): 在Keras模型上会被自动调用，History对象即为fit方法的返回值

callbacks.ProgbarLogger(): 将metrics指定的监视指标输出到标准输出上

callbacks.ModelCheckpoint(filepath ...): 在每个epoch后保存模型到filepath

callbacks.EarlyStopping(...): 当监测值不再改善时，该回调函数将中止训练

callbacks.RemoteMonitor(root='http://localhost:9000'):向服务器发送事件流，该回调函数需要requests库

callbacks.TensorBoard(log\_dir='./logs', ...): 可视化的展示器

callbacks.ReduceLROnPlateau(monitor='val\_loss', factor=0.1,...):当评价指标不在提升时，减少学习率

callbacks.CSVLogger(filename, separator=',', append=False):将epoch的训练结果保存在csv文件中

编写自己的回调函数

class LossHistory(keras.callbacks.Callback):

def on\_train\_begin(self, logs={}):

self.losses = []

def on\_batch\_end(self, batch, logs={}):

self.losses.append(logs.get('loss'))

history = LossHistory()

model.fit(X\_train, Y\_train, batch\_size=128, epochs=20, verbose=0, callbacks=[history])

print history.losses

keras.utils提供的使得方法

x\_data = keras.utils.io\_utils.HDF5Matrix('input/file.hdf5', 'data')

model.predict(x\_data)

keras.utils.to\_categorical(y, num\_classes=None)

keras.utils.normalize(x, axis=-1, order=2)

x：待规范化的数据

axis: 规范化的轴

order：规范化方法，如2为L2范数

#### Keras后端

Keras是一个模型级的库，提供了快速构建深度学习网络的模块。Keras并不处理如张量乘法、卷积等底层操作。这些操作依赖于某种特定的、优化良好的张量操作库。Keras依赖于处理张量的库就称为“后端引擎”。Keras提供了两种后端引擎Theano/Tensorflow

Keras的配置文件$HOME/.keras/keras.json

文件的默认配置如下：

{

"image\_data\_format": "channels\_last",

"epsilon": 1e-07,

"floatx": "float32",

"backend": "tensorflow"

}

也可以通过定义环境变量KERAS\_BACKEND来覆盖上面配置文件中定义的后端

使用抽象的Keras后端来编写代码

from keras import backend as K

定义tensor

#tf.placeholder() ，T.matrix()，T.tensor3()

input = K.placeholder(shape=(2, 4, 5))

#共享变量（shared），等价于tf.variable()或 theano.shared()

val = np.random.random((3, 4, 5))

var = K.variable(value=val)

var = K.zeros(shape=(3, 4, 5)) # all-zeros variable:

var = K.ones(shape=(3, 4, 5)) # all-ones

K.eye(size, dtype='float32', name=None)

大多数你需要的张量操作都可以通过统一的Keras后端接口完成，而不关心具体执行这些操作的是Theano还是TensorFlow

a = b + c \* K.abs(d)

c = K.dot(a, K.transpose(b))

a = K.sum(b, axis=2)

a = K.softmax(b)

eps = K.epsilon()

K.set\_epsilon(1e-05)

K.floatx()

K.set\_floatx('float16')

data\_format = K.image\_data\_format() #‘channels\_last’或‘channels\_first’

K.set\_image\_data\_format(data\_format)

K.is\_keras\_tensor(input)

zeros\_like, ones\_like, random\_uniform\_variable, max, min, sum, prod, cumsum, cumprod, var, std, mean, any, all, argmax, square, abs, sqrt, exp, log, logsumexp, round, sign, pow, clip, equal, sin, cos, concatenate, reshape, permute\_dimensions, resize\_images, resize\_volumes, repeat\_elements, repeat, arange, tile, batch\_flatten, expand\_dims, squeeze,random\_normal, random\_uniform,random\_binomial,truncated\_normall

normalize\_batch\_in\_training(): 对一个batch数据先计算其均值和方差，然后再进行batch\_normalization

batch\_normalization(): output = (x-mean)/(sqrt(var)+epsilon)\*gamma+beta

one\_hot(indices, nb\_classes): 输出为(n+1)维的one-hot编码

get\_value(x): 以Numpy array的形式返回张量的值

gradients(loss, variables): 返回loss函数关于variables的梯度，variables为张量变量的列表

relu(x, alpha=0.0, max\_value=None)

elu(x, alpha=1.0)

softmax(x)

softplus(x)

softsign(x)

sigmoid(x)

hard\_sigmoid(x)

tanh(x)

categorical\_crossentropy(output, target, from\_logits=False): 计算输出张量和目标张量的Categorical crossentropy（类别交叉熵）

l2\_normalize(x, axis)

dropout(x, level, seed=None)

conv1d(...)

conv2d(...)

deconv2d(...)

conv3d(...)

pool2d(...)

pool3d(...)

bias\_add(x, bias, data\_format=None)

### workflow

Load Data.

Define Model.

Compile Model.

Fit Model.

Evaluate Model.

Tie It All Together.

### examplehttp://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks

#### image segmentation

lung segmentation

<https://blog.altoros.com/experimenting-with-deep-neural-networks-for-x-ray-image-segmentation.html>

Melanoma (lesion analysis)

//appsrv.cse.cuhk.edu.hk/~lqyu/skin/

Dermoscopy 皮肤镜

Deep Residual Learning （DRN）

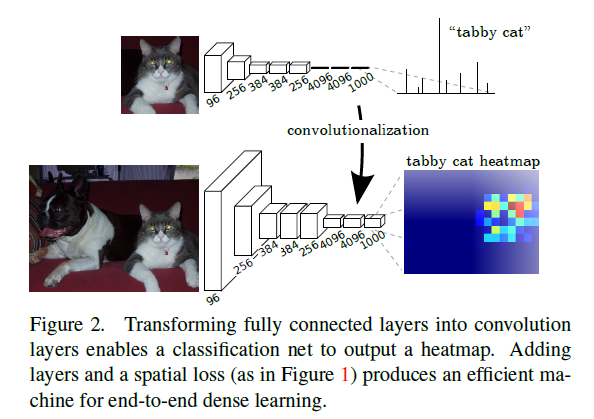
//blog.csdn.net/yaoxingfu72/article/details/50764087

#### Image denoising

### Model<https://isic-archive.com/>

#### fully convolutional network

Contemporary classification networks (AlexNet, VGGnet, GoogleNet) -> FCN



blog.csdn.net/tangwei2014/article/details/46882257

1）如何做pixelwise的prediction？

传统的网络是subsampling的，对应的输出尺寸会降低，要想做pixelwise prediction，必须保证输出尺寸。

解决办法：

（1）对传统网络如AlexNet，**VGG等的最后全连接层变成卷积层**。

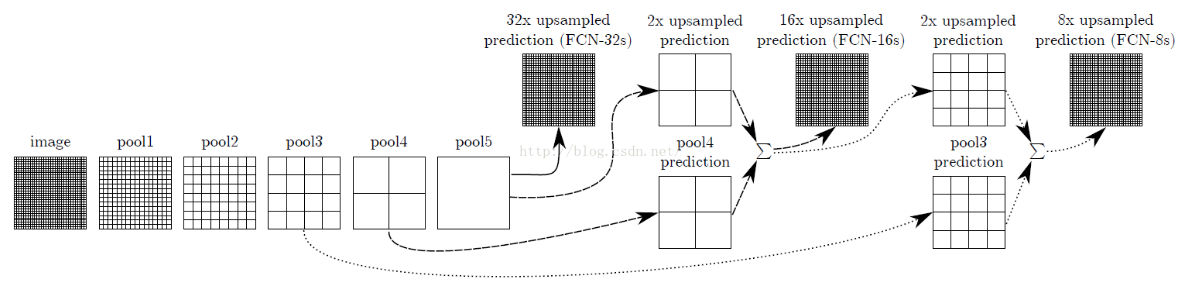
例如VGG16中第一个全连接层是25088x4096的，将之解释为512x7x7x4096的卷积核，则如果在一个更大的输入图像上进行卷积操作（上图的下半部分），原来输出4096维feature的节点处（上图的上半部分），就会输出一个coarse feature map。

（2）加 In-network upsampling layer。

**对中间得到的feature map做bilinear上采样**，就是反卷积层。实现把conv的前传和反传过程对调一下即可。

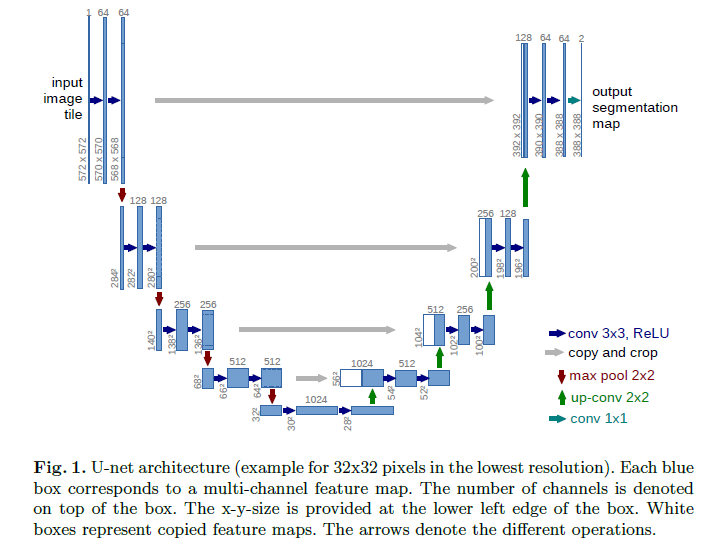
2）如何refine，得到更好的结果？

upsampling中步长是32，输入为3x500x500的时候，输出是544x544，边缘很不好，并且limit the scale of detail of the upsampling output。



采用skip layer的方法，在浅层处减小upsampling的步长，得到的fine layer 和 高层得到的coarse layer做融合，然后再upsampling得到输出。这种做法兼顾local和global信息，即文中说的combining what and where，取得了不错的效果提升。FCN-32s为59.4，FCN-16s提升到了62.4，FCN-8s提升到62.7。可以看出效果还是很明显的。

#### Unet



u-net: u-shaped architecture

Build upon “fully convolutional network”

Merge contracting path and upsampling path

Data augmentation:

elastic deformation (which is the most common variation in tissue)

shift, rotation invariance

Weighted loss: give some pixels more importance in the training

[**https://github.com/iamrosmarin/BSc\_Thesis\_Skin\_Lesion\_Detection**](https://github.com/iamrosmarin/BSc_Thesis_Skin_Lesion_Detection)

## tensorflow

tensor (n维数组)　+ 计算图

充分利用CPU and GPU

Pycharm tensorflow ImportError but works fine with Terminal?

open PyCharm from the command line and everything works now

# Machine learning

## MLP (multi-layer perception)

Sigmoid + Sigmoid + mse

batch Gradient decent, mini-batch gradient decent, SGD

<http://iamtrask.github.io/2015/07/12/basic-python-network/> <http://iamtrask.github.io/2015/07/27/python-network-part2/>

<https://iamtrask.github.io/2015/07/27/python-network-part2/>

drops out<http://iamtrask.github.io/2015/07/28/dropout/>



The line represents the error the network generates for every value of a particular weight, The balls in the picture signify various weights, The ball's initial positions are randomly generated, If two balls randomly start in the same colored zone, they will converge to the same point. This makes them redundant! They're wasting computation and memory! This is exactly what happens in neural networks.

Why Dropout: Dropout helps prevent weights from converging to identical positions. It does this by **randomly turning nodes off when forward propagating**. It then **back-propagates with all the nodes turned on**

if(do\_dropout):

layer\_1 \*= np.random.binomial([np.ones((len(X),hidden\_dim))],1-dropout\_percent)[0]

\* (1.0/(1-dropout\_percent))

if you're turning off half of your hidden layer, you want to double the values that ARE pushing forward so that the output compensates correctly

Dropout during training. not on your testing dataset

Tanh + softmax + cross-entropy loss

<http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/>

动量项

<http://blog.csdn.net/bvl10101111/article/details/54973284>

batch normalization

xk = xk – E[xk] / sqrt(Var[xk])

yk = rkxk + betak

对mini-batch数据集，沿着第k维进行batch normalization(公式如上)

优点：

improves gradient flow through the networks

allows higher learning rates

reduces the strong dependence on initialization

acts as a form of regularization, slightly reduces the need for dropout

role of bias in Neural Networks

sigmoid(wx+b), w改变的是sigmoid曲线的steepness, b改变的是将x平移

# deep learning

## Introduction to Deep Learning Algorithms

Traditional feedforward neural networks can be considered to have depth equal to the number of layers (i.e. the number of hidden layers plus 1, for the output layer). Support Vector Machines (SVMs) have depth 2 (one for the kernel outputs or for the feature space, and one for the linear combination producing the output).

Motivations for Deep Architectures

1. Insufficient depth can hurt

Depth 2 is enough in many cases (e.g. logical gates, formal [threshold] neurons, sigmoid-neurons, Radial Basis Function [RBF] units like in SVMs) to represent any function with a given target accuracy. But this may come with a price: that the required number of nodes in the graph may grow very large.

1. **The brain has a deep architecture**
2. Cognitive processes seem deep

Humans organize their ideas and concepts hierarchically.

**Humans first learn simpler concepts and then compose them to represent more abstract ones.**

Engineers break-up solutions into multiple levels of abstraction and processing

Deep Learning Tutorial

Training set

Validation set (perform model selection and hyper-parameter selection)

Test set (evaluate the final generalization error and compare different algorithms)

## 一天搞懂深度学习 –李宏毅http://www.slideshare.net/tw\_dsconf/ss-62245351

### NN



A two layers of logic gates can represent **any Boolean function**

A hidden layer network can represent **any continuous function**

Using multiple layers of neurons to represent some functions are much simpler -> less parameters, less data?

直接分类长发女，短发女，长发男，短发男，很困难，因为训练集少。但若先分类男女？长短头发？由这两个basic classifier再组合进一步分类，更容易

Learning Rates

小梯度，平坦，则大步伐；大梯度，陡峭，则小步伐

Weight Decay

Our brain prunes out the useless link between neurons, doing the same thing to machine’s brain improves the performance

Dropout直观理解：

### CNN

Why CNN for Image?

Some patterns are much smaller than the whole image

Subsmapling the piexls will not change the object



### RNN (Recurrent Neural Network)

## CS231n Convolutional Neural Networks for Visual Recognition

### NN (Neural Network)

网络权值的解释？W = 模板

Interpretation of **linear classifiers as template matching**. Another interpretation for the weights W is that each row of W corresponds to a template (or sometimes also called a prototype) for one of the classes. The score of each class for an image is then obtained by comparing each template with the image using an inner product (or dot product) one by one to find the one that “fits” best. With this terminology, the linear classifier is doing template matching, where the templates are learned. Another way to think of it is that we are still effectively doing Nearest Neighbor, but instead of having thousands of training images we are only using a single image per class (although we will learn it, and it does not necessarily have to be one of the images in the training set), and we use the (negative) inner product as the distance instead of the L1 or L2 distance.

for example: the ship template (W) contains a lot of blue pixels as expected. This template will therefore give a high score once it is matched against images of ships on the ocean with an inner product.

Additionally, note that the horse template seems to contain a two-headed horse, which is due to both left and right facing horses in the dataset. The linear classifier merges these two modes of horses in the data into a single template. Similarly, **the car linear classifier seems to have merged several modes into a single template which has to identify cars from all sides, and of all colors**. In particular, this template ended up being red, which hints that there are more red cars in the CIFAR-10 dataset than of any other color. The linear classifier is too weak to properly account for different-colored cars, but as we will see later neural networks will allow us to perform this task. Looking ahead a bit, a **neural network will be able to develop intermediate neurons in its hidden layers that could detect specific car types (e.g. green car facing left, blue car facing front, etc.), and neurons on the next layer could combine these into a more accurate car** score through a weighted sum of the individual car detectors.

First-layer Visualizations



Examples of visualized weights for the first layer of a neural network. Left: **Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty**. Right: Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

Multi-layer Visualization

Layer1 focus on **local domain feature**

Layer2 combine layer1’s output (as basic classifier) = combine local feature to **multi-features**



linear mapping -> Neural Networks -> Convolutional Neural Networks

score function: mapping the raw image pixels to class scores (e.g. a linear function)

loss(cost) function: measured the quality of a particular set of parameters based on how well the induced scores agreed with the ground truth labels in the training data (e.g. Softmax/SVM).

SVM cost function is an example of a convex function --- convex optimization

Neural Networks cost functions will become non-convex

如何优化？

Optimization: the process of finding the set of parameters W that minimize the loss function.

1. Random search (bad idea)

W = np.random.randn(10, 3073) \* 0.0001 # generate random parameters

1. Random Local Search

Core idea: iterative refinement, refining a specific set of weights W to be slightly better is significantly less difficult

Wtry = W + np.random.randn(10, 3073) \* step\_size

1. Following the Gradient

weights += - step\_size \* weights\_grad # perform parameter update

The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step

choosing the step size (also called the learning rate) will become one of the most important (and most headache-inducing) hyper parameter settings in training a neural network

如何计算梯度？

Computing the gradient: numerical gradient and analytic gradient (e.g. **chain rule == backpropagation**)

Backpropagation allow us to efficiently optimize relatively arbitrary loss functions that express all kinds of Neural Networks, including Convolutional Neural Networks.

Batch梯度下降法(BGD)：it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update

Mini-batch 梯度下降法(MGD)：compute the gradient over batches of the training data

假定120万张图像，由1000 labels组成，那么120万张的平均data loss与1000张等值，the gradient from a mini-batch is a good approximation of the gradient of the full objective (cost)

The size of the mini-batch is a hyper parameter but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

Stochastic Gradient Descent (SGD) : on-line gradient descent

Tips:

if loss barely changing means Learning rate is probably too low

mini-batch size: 256

step size: 10\*\*(-5) (ideally, h->0)

2% improvement: train multiple independent models, at test time average their results

Mini-batch SGD

Loop:

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

Regularization (dropout)

Randomly set some neurons to zero in the forward pass



解释1：



解释2:

Dropout is training a large ensemble of models (that share parameters). Each binary mask is one model gets trained on only one data point

Ideally, want to integrate out all the noise.

### CNN (Convolutional Neural Networks / ConvNets)

ConvNet architectures:

Images -> Convolutional Layer -> Pooling Layer -> Fully-Connected Layer -> output labels

ConvNet architectures make the explicit assumption that the inputs are images

Regular Neural Nets don’t scale well to full images, for example, image 200\*200\*3 -> 120,000 weights

full connectivity is wasteful and **the huge number of parameters would quickly lead to overfitting**.

3D volumes of neurons: unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth.

Intuitively, the network will **learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer**, or eventually entire honeycomb or wheel-like patterns on higher layers of the network, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.



For example, if the first Convolutional Layer takes as input the raw image, then different neurons may activate in presence of various oriented edged, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a depth column (some people also prefer the term fibre).

Real-world example

the input volume size (W1 \* H1 \* D1): 227\*227\*3

the receptive field size of the Conv Layer neurons (F): 11

the number of filters (K): 96

the stride with which they are applied (S): 4

the amount of zero padding used (P) on the border: 0

the spatial size of the output volume (W2 \* H2 \* D2): 55\*55\*96

W2 = **(W1-F+2P)/S+1** = 55

H2 = (H1−F+2P)/S+1 = 55

D2 = K = 96

With parameter sharing, each filter has F\*F\*D1 (11\*11\*3 = 363) weights, **total weights (F\*F\*D1)\*K (363x96) and K biases**

理论上：参数有55x55x96 \* 11\*11\*3 多个

假设：Notice that **the parameter sharing assumption** is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images.

假设不一定对：Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than another. One practical example is when the input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a Locally-Connected Layer.

若假设对，则图像空间共用同一个filter（理由如上）,则参数：96 \* 11\*11\*3

Backpropagation: The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters).

all 55\*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Pooling Layer



reduce the amount of parameters and computation in the network, and hence to also control overfitting

how? filters of size 2x2 applied with a stride of 2 down samples every depth slice

input volumn size: W1\*H1\*D1

filter (F): 2\*2

stride (S): 2

output volumn size: W2\*H2\*D2

W2 = **(W1-F)/S + 1**

H2 = (H1-F)/S + 1

D2 = D1

general pooling: max, average, L2-norm pooling

Backpropagation: Recall from the backpropagation chapter that **the backward pass for a max(x, y) operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass**. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation.

Fully-connected layer

Converting FC layers to CONV layers

input volume of size: 7\*7\*512

F = 7

P = 0

S = 1

K = 4096

output: 1\*1\*4096

Layer Patterns

INPUT -> [[CONV -> RELU]\*N -> POOL?]\*M -> [FC -> RELU]\*K -> FC

usually

**0 <= N <= 3**

**0 <= M**

**0 <= K < 3**

INPUT -> FC, implements a linear classifier. Here N = M = K = 0.

INPUT -> CONV -> RELU -> FC

INPUT -> [CONV -> RELU -> POOL]\*2 -> FC -> RELU -> FC. Here we see that there is a single CONV layer between every POOL layer.

INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]\*3 -> [FC -> RELU]\*2 -> FC Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because **multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation**

多个小filter级别优于大filter

Suppose that you stack three 3x3 CONV layers on top of each other (with non-linearities in between, of course). In this arrangement, **each neuron on the first CONV layer has a 3x3 view of the input volume**. **A neuron on the second CONV layer has a 3x3 view of the first CONV layer, and hence by extension a 5x5 view of the input volume**. Similarly, a neuron on the third CONV layer has a 3x3 view of the 2nd CONV layer, and hence a 7x7 view of the input volume. Suppose that instead of these three layers of 3x3 CONV, we only wanted to use a single CONV layer with 7x7 receptive fields.

3个3\*3filter等效于7\*7filter，但有如下优点

First, the neurons would be computing a linear function over the input, while the three **stacks of CONV layers contain non-linearities that make their features more expressive**.

Second, if we suppose that all the volumes have C channels, then it can be seen that the single 7x7 CONV layer would contain C×(7×7×C)=49C2parameters, while the three 3x3 CONV layers would only contain 3×(C×(3×3×C))=27C2 parameters.

Intuitively, **stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters**

INPUT: [224x224x3] memory: 224\*224\*3=150K weights: 0

CONV3-64: [224x224x64] memory: 224\*224\*64=3.2M weights: (3\*3\*3)\*64 = 1,728

CONV3-64: [224x224x64] memory: 224\*224\*64=3.2M weights: (3\*3\*64)\*64 = 36,864

POOL2: [112x112x64] memory: 112\*112\*64=800K weights: 0

CONV3-128: [112x112x128] memory: 112\*112\*128=1.6M weights: (3\*3\*64)\*128 = 73,728

CONV3-128: [112x112x128] memory: 112\*112\*128=1.6M weights: (3\*3\*128)\*128 = 147,456

POOL2: [56x56x128] memory: 56\*56\*128=400K weights: 0

CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*128)\*256 = 294,912

CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*256)\*256 = 589,824

CONV3-256: [56x56x256] memory: 56\*56\*256=800K weights: (3\*3\*256)\*256 = 589,824

POOL2: [28x28x256] memory: 28\*28\*256=200K weights: 0

CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*256)\*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*512)\*512 = 2,359,296

CONV3-512: [28x28x512] memory: 28\*28\*512=400K weights: (3\*3\*512)\*512 = 2,359,296

POOL2: [14x14x512] memory: 14\*14\*512=100K weights: 0

CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14\*14\*512=100K weights: (3\*3\*512)\*512 = 2,359,296

POOL2: [7x7x512] memory: 7\*7\*512=25K weights: 0

FC: [1x1x4096] memory: 4096 weights: 7\*7\*512\*4096 = 102,760,448

FC: [1x1x4096] memory: 4096 weights: 4096\*4096 = 16,777,216

FC: [1x1x1000] memory: 1000 weights: 4096\*1000 = 4,096,000

TOTAL memory: 24M \* 4 bytes ~= 93MB / image (only forward! ~\*2 for bwd)

TOTAL params: 138M parameters

## DeepLearning

许多数学对象可通过将它们分解成多个组成部分，如整数可以质因数分解，矩阵可以奇异值分解，实对称矩阵可以特征分解