Contents

[Contents 1](#_Toc502748735)

[Machine learning 2](#_Toc502748736)

[MLP (multi-layer perception) 2](#_Toc502748737)

[Deep learning framework 3](#_Toc502748738)

[Keras 5](#_Toc502748739)

[安装 6](#_Toc502748740)

[概念 7](#_Toc502748741)

[workflow 28](#_Toc502748742)

[example 28](#_Toc502748743)

[数据预处理 29](#_Toc502748744)

[tensorflow 30](#_Toc502748745)

[deep learning 39](#_Toc502748746)

[Introduction to Deep Learning Algorithms 39](#_Toc502748747)

[一天搞懂深度学习 –李宏毅 40](#_Toc502748748)

[NN 40](#_Toc502748749)

[CNN 42](#_Toc502748750)

[RNN (Recurrent Neural Network) 42](#_Toc502748751)

[CS231n Convolutional Neural Networks for Visual Recognition 42](#_Toc502748752)

[NN (Neural Network) 42](#_Toc502748753)

[CNN (Convolutional Neural Networks / ConvNets) 46](#_Toc502748754)

[DeepLearning 50](#_Toc502748755)

[矩阵论 51](#_Toc502748756)

[概率与信息论 51](#_Toc502748757)

[数值计算 53](#_Toc502748758)

[机器学习基础 54](#_Toc502748759)

[深度前馈网络 58](#_Toc502748760)

[深度学习中的正则化 60](#_Toc502748761)

[深度模型中的优化 63](#_Toc502748762)

[卷积网络 63](#_Toc502748763)

[模型 63](#_Toc502748764)

[LeNet 64](#_Toc502748765)

[AlexNet 65](#_Toc502748766)

[VggNet-19 67](#_Toc502748767)

[GoogLeNet 67](#_Toc502748768)

[ResNET 69](#_Toc502748769)

[fully convolutional network 71](#_Toc502748770)

[SegNet 74](#_Toc502748771)

[Unet 75](#_Toc502748772)

[对抗样本和对抗网络 76](#_Toc502748773)

# Machine learning

入门

From The Official Blog of Kaggle.com

blog.kaggle.com/2016/07/21/approaching-almost-any-machine-learning-problem-abhishek-thakur/

## MLP (multi-layer perception)

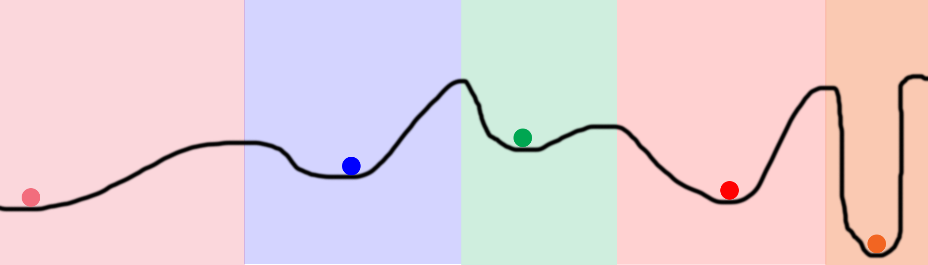
Sigmoid + Sigmoid + mse

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 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

若内存足够大，一次将所有训练数据导入内存

model.fit(X\_train, Y\_train, batch\_size=batch\_size, nb\_epoch=12)

若CPU内存不大，无法一次性将所有训练数据导入，怎么办？

批次导入训练数据

存在多种方案：

方案一：先导入subset1, 训练12次，然后导入subset2,训练12次，...直至遍历所有训练集

方案二：先导入subset1, 训练1次，然后导入subset2, 训练1次，...遍历所有训练集，然后重复以上12次

以上是对mnist数据集的比较结果：建议采用方案二

0.989 total, 12 epoch 397s

0.986 11\*subset, 12 epoch / subset 396s

0.989 12 \* (11\*subset + 1 epoch / subset) 393s

keras 代码

from psutil import virtual\_memory

mem = virtual\_memory()

free\_memory = ratio \* float(mem.total) / 1024 \*\* 3

case\_num\_memory = f(free\_memory, case\_size)

subset\_num = ceil(all\_case\_num/subset\_num)

for j in range(epoches):

for i in range(subset\_num):

history = model.fit(train.subset(i), batch\_size=batch\_size, 1)

tensorflow代码

batch\_num = ceil(all\_case\_num/batch\_size)

for j in range(epoches):

for i in range(batch\_num):

batch = train.next\_batch(batch\_size)

sess.run([...], feed\_dict={X:batch[0], Y:batch[1]}

## 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**(...)

**predict**(…)

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

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

#### 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

**权重初始化**

**from keras import initializers**

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')

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)

**Key points:**

* 数据集

A batch generally approximates the distribution of the input data better than a single input. The larger the batch, the better the approximation

大数据集，内存放不下，如何处理？

model.train\_on\_batch(x, y)

model.test\_on\_batch(x, y)

model.fit\_generator(data\_generator, steps\_per\_epoch, epochs)

小数据集，如何解决overfitting?

blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

模型太大，不仅学到共性特征，也学到训练集独有的特征，可以添加regular, dropout等解决

Data augmentation，使model训练时见到没有同样的case

Transfer 学习， 用相近的更大规模训练，然后freeze 低层，只对top层进行训练

开发阶段，如何避免性能改变是由于随机改变，还是模型提升？

keras.io/getting-started/faq/

import os

os.environ['PYTHONHASHSEED'] = '0'

np.random.seed(42)

rn.seed(12345)

* 模型 f(input\_shape, n\_class)

Model 需要指定输入维数（不需要包含batch\_size）, 由input tensor and output tensor定义

可以将model视为layers，输入tensor, 输出tensor

input\_shape = (w, h, channel)

model = Model(Input(input\_shape), output)

Y\_predict = model(X\_train)

注意，同样地，所有层，不考虑batch\_size

比如

h1 = Dense(32, input\_shape=(16, )) 输入(\*, 16), 输出(\*, 32)

c1 = Conv2D(64, 3, 3)(pool1) 若pool1(\*, 32, 32, 3), 输出(\*, 32, 32, 64)

f1 = Flatten()(c1) 输出(\*, 32\*32\*64)

r1 = Reshape((32, 32, 64))(f1) 输出(\*, 32, 32, 64)

Permute((1, 3))(r1) 输出(\*, 64, 32, 32)

RepeatVector(3)(h1) 输出(\*, 3, 32)

自定义层

方法一：若无状态操作，采用layers.core.Lambda

若输出维数不变：model.add(Lambda x: x\*\*2)

若输出维数改变：

def antirectifier(x):

pos = K.relu(x)

neg = K.relu(-x)

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

def output\_shape(input\_shape):

shape = list(input\_shape)

shape[-1] \*= 2

return tuple(shape)

model.add(Lambda(antirectifier, output\_shape=output\_shape))

方法二：若有状态操作，如含权重，需要继承Layer

**from keras import backend as K**

**from keras.engine.topology import Layer**

import numpy as np

#阅读源代码

class Conv2D(**Layer**):

def \_\_init\_\_(self, kernel\_size=(3,3), filters=32, \*\*kwargs):

self.kernel\_size = kernel\_size

self.filters = filters

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

#**定义权重**的方法 # Create a trainable weight variable for this layer.

**def build(self, input\_shape):**

channels\_axis = -1 if self.data\_format==’channels\_last’ else 1

input\_dim = input\_shape[channels\_axis]

kernel\_shape = self.kernel\_size + (input\_dim, self.filters)

self.kernel = self.add\_weight(shape=kernel\_shape, initializer='uniform')

super(MyLayer, self).build(input\_shape)

**#定义层功能的方法**

**def call(self, inputs):**

return K.con2d(inputs, self.kernel)

#**指定shape**变化的方法

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

return (input\_shape[0], input\_shape[1], self.filters )

保存/导入模型Saving/loading whole models (architecture + weights + optimizer state)

from keras.models import load\_model

model.save('my\_model.h5') # creates a HDF5 file 'my\_model.h5'

del model # deletes the existing model

# returns a compiled model

# identical to the previous one

model = load\_model('my\_model.h5')

* 训练

若指标没有随训练下降，停止学习

from keras.callbacks import EarlyStopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=2)

model.fit(x, y, validation\_split=0.2, callbacks=[early\_stopping])

* 正则化：

mnist 60000训练集，加了reg, acc从99.11 → 99.12

Conv2D(32, (5, 5), activation='relu',

kernel\_regularizer=l2(weight\_decay),

bias\_regularizer=l2(weight\_decay),

activity\_regularizer=l2(weight\_decay))

* 不平衡类：

mnist 60000训练集，加了class\_weight, acc从99.12 → 99.19

class\_weight = {0 : 1., 1: 50., 2: 2.} 对应类权重

model.fit(X\_train, Y\_train， class\_weight = class\_weight)

计算类权重，越少类权重越大

from sklearn.utils.class\_weight import compute\_class\_weight

class\_weight = compute\_class\_weight('balanced', np.unique(y\_train), y\_train)

model.fit(X\_train, y\_train, class\_weight=class\_weight)

熵作为类权重

log(1 / yi) 其中yi ＝ count / (mu\*total), mu = 0.15

# labels\_dict : {ind\_label: count\_label}

# mu : parameter to tune

def create\_class\_weight(labels\_dict, mu=0.15):

total = np.sum(labels\_dict.values())

class\_weight = {}

for key in labels\_dict.keys():

score = math.log(mu\*total/float(labels\_dict[key]))

class\_weight[key] = score if score > 1.0 else 1.0

return class\_weight

* 损失函数 f(target, output)

**input have shape [batch\_size, w, h, 1]**

**output have shape [batch\_size, 1]**

target = K.reshape(target, [-1, w\*h\*1])

output = K.reshape(output, [-1, w\*h\*1])

dice\_coef

intersection = K.sum(y\_true \* y\_pred, axis=-1)

return (2. \* intersection + smooth) /

(K.sum(y\_true, axis=-1) + K.sum(y\_pred, axis=-1) + smooth)

binary\_cross\_entropy

pos\_weight = 2

loss = pos\_weight \* y\_true \* K.log(y\_pred+K.epsilon()) + (1-y\_true)\*K.log(1-y\_pred+K.epsilon())

return -K.mean(loss, axis=-1)

keras的损失函数binary\_crossentropy支持[batch\_size, w, h, 1], 但最后一层需为sigmoid激活函数

model.compile(loss=binary\_crossentropy, metrics=[dice\_coef])

但是class\_weight不支持3D以上, 自定义可以实现此功能

输出为 [batch\_size, 1], 每个元素表示case的loss, keras会求平均

若输出为标量，则表示batch\_size case的平均loss

dice\_coef

intersection = K.sum(y\_true \* y\_pred)

return 2. \* intersection / (K.sum(y\_true) + K.sum(y\_pred) )

categorical\_crossentropy

若logits and y have same shape (32, 512, 512, 10)

则class\_weights have shape (1, 10)

flat\_logits = tf.reshape(logits, [-1, n\_class]) (32\*512\*512, 10)

flat\_labels = tf.reshape(y, [-1, n\_class])

class\_weights = tf.constant(np.array(class\_weights, dtype=np.float32))

weight\_map = tf.multiply(flat\_labels, class\_weights) (32\*512\*512, 10)

weight\_map = tf.reduce\_sum(weight\_map, axis=1) (32\*512\*512, 1)

(32\*512\*512, 1) ← (32, 512, 512, 10) , (32, 512, 512, 10)

loss\_map = tf.nn.softmax\_cross\_entropy\_with\_logits(flat\_logits, flat\_labels)

weighted\_loss = tf.multiply(loss\_map, weight\_map) (32\*512\*512, 1)

loss = tf.reduce\_mean(weighted\_loss) scalar

* Metrics f(y\_true, y\_pred)

**#(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])

### 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

图像分割方法文献回顾：

nicolovaligi.com/deep-learning-models-semantic-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

### 数据预处理

图像以较小的边缩放至[256, 480], 这样便于scale augmentation,然后从中随机裁出224\*224

水平翻转

标准颜色扩充

减均值

优化

Initialization

Initialization under ReLU activation --kaiming he

## tensorflow

TensorFlow + TensorBoard + TensorFlow Serving

TensorFlow =

TensorFlow Core (low level API) +

tf.contrib.learn (high-level API)

设计目标：在多台计算机以及单机多CPU,单机多GPU环境中具有良好的可伸缩性

定义计算图 ＋ 运行计算图（在数据上）

**计算图本质是一组链接在一起的函数**，每个函数都会将其输出传递给0,1,多个位于这个级联链上的其他函数

计算图（数据流图） ＝ 节点（对数据所做的运算或输入操作）＋边（tensor)

A computational graph is a series of TensorFlow operations arranged into a graph of nodes

To actually evaluate the nodes, we must run the computational graph within a session. A session encapsulates the control and state of the TensorFlow runtime.

一个 TensorFlow 图描述了计算的过程. 为了进行计算, 图必须在 会话 里被启动. 会话 将图的 op 分发到诸如 CPU 或 GPU 之类的 设备 上, 同时提供执行 op 的方法. 这些方法执行后, 将产生的 tensor 返回

当TensorFlow库被加载时，它会自动创建一个Graph对象，并将其作为默认的数据流图。因此在Graph.as\_default()上下文管理器之外定义的任何Op, Tensor对象都会自动放置在默认的数据流图中。

在大多数TensorFlow程序中，只使用默认数据流图就足够了。然而，如果需要定义多个相互之间不存在依赖关系的模型，则创建多个Graph对象十分有用。

g1 = tf.get\_default\_graph()

g2 = tf.Graph()

with g1.as\_default():

定义g1的Op 和张量等

with g2.as\_default():

定义g2的Op和张量等

Tensor

一个 op 获得 0 个或多个 Tensor, 执行计算, 产生 0 个或多个 Tensor. 每个 Tensor 是一个类型化的多维数组

**TensorFlow Op可接收标准Python数据类型，如整数或字符串，并将它们自动转化为张量，返回的 tensor 是 numpy ndarray 对象**

**经典做法是借助Numpy数组手工定义Tensor对象**,TensorFlow的数据类型是基于Numpy的数据类型的。实际上，语句np.int32==tf.int32。任何NumPy数组都可以传递给TensorFlow Op

t\_0 = np.array(50, dtype=np.int32) 0阶张量 即标量

t\_1 = np.array([1., 2.], dtype=np.float32) 1阶张量即向量

t\_2 = np.array([ 2阶张量即矩阵

[1, 2, 3],

[4, 5, 6]

], dtype=np.int32)

除了能够将张量的每一维指定为固定长度，也可以将None作为某一维的值，使该疑张量具有可变长度。

s\_0 = () 0阶张量的形状

s\_1 = (3) 1阶张量的形状

s\_2 = (3, 2)

**s\_2\_flex = (None, 2) 行数任意，列数为3的矩阵形状**

s\_any = None 形状任意的张量

shape = tf.shape(tensor1)

sess.run(shape) 求张量tensor1的shape

节点Op

a = np.array([2, 3], dtype=np.int32) tensor

b = tf.constant([5, 3], name=”input\_b”) 节点，指向常量操作输出Tensor句柄

**c = tf.add(a, b) 变量c为指向该Op输出的Tensor对象的句柄**

实际上，存在对应的运算符重载，张量逐元素运算+-\*/, //, %, \*\*, < <= > >= & | ^

c = a + b 运算符重载

如何确保计算机中对必要的节点执行运算，而无需手工指定？

答案是：利用节点之间的依赖关系

可对数据流中的任意Op使用run()函数。当将某个Op传入sess.run()时，本质上是通过TensorFlow“这里有一个节点，我希望得到它的输出，请执行所有必要的运算来求取这个节点的输出”

a = tf.add(2, 3)

b = tf.mul(a, 3)

with tf.Session() as sess: 等价于tf.Session(graph=tf.get\_default\_graph())

sess.run(tf.initialize\_all\_variables())

sess.run(fetches=[b], feed\_dct={a: 15})

fetches参数接收任意的数据流图元素(Op or Tensor对象），后者指定了用户希望执行的对象。若请求的是Tensor对象，则run()输出的是Numpy数组； 若请求的是Op，则输出为None

feed\_dict用于覆盖数据流图中的Tensor对象值，**字典键为Tensor对象的句柄**

sess.run([b], feed\_dct={a: 15})

例子中取张量b， TensorFlow便会得到通知，Session对象应当找到为计算b的值所需的全部节点，顺序执行这些节点，然后b的值输出

sess.run(tf.initialize\_all\_variables()) 执行初始化Variable对象所需的计算，返回值为None

summary.FileWriter 对象保存来自数据流图的数据和概括流计量，第一个参数是数据流图的描述在磁盘中的存放路径。第二个参数是要追踪的数据流图

writer = tf.summary.FileWriter(‘./my\_graph’, sess.graph)

一量执行完以上代码，便可启动TensorBoard

$ tensorboard –-logdir=”.my\_graph”

localhost:6006会显示利用Writer对象要求TensorFlow所保存的信息。

利用占位节点添加输入

a = tf.placeholder(tf.int32, shape=[2])

b = tf.reduce\_sum(a)

sess.run(b, feed\_dict={a: np.array([5, 3], dtype=np.int32)})

变量

Tensor对象和Op对象都是不可变的，如何维护图执行过程中的状态信息？

答案是：变量(Variables for more details),包含了在对Session.run()多次调用中可持久化的可变张量值

通常会将一个统计模型中的参数表示为一组变量. 例如, 你可以将一个神经网络的权重作为某个变量存储在一个 tensor 中. 在训练过程中, 通过重复运行训练图, 更新这个 tensor.

weight = tf.Variable(3)

Variables对象通常需要初值，TensorFlow提供辅助Op

weights = tf.Varaible(tf.zeros([2, 2]))

tf.ones([6])

tf.random\_uniform([3, 3, 3], minval=0, maxval=10)

tf.random\_normal([3, 3, 3], mean=0.0, stddev=2.0)

#创建[mean-2std, mean+2std]

tf.truncated\_normal([2, 2], mean=5.0, stddev=1.0)

为使用Variable对象，需要采取一些额外的步骤，必须在一个Session对象内对Variable对象进行初始化。这样会使Session对象开始追踪这个Variable对象的值的变化

sess.run(tf.initialize\_all\_variables())

修改Variable

weights\_update = weights.assign(weights \* 2)

sess.run(weights) 因为还未运行操作weights.assign，所以weights未改变

sess.run(weights\_update)

运行操作weights.assign, weights and weights\_update均改变

由于不同Session对象会各自独立地维护Variable对象的值，因此每个Session对象都拥有自己的，在Graph对象中定义的Variable对象的当前值

若将所有Variable对象的值重置为初始值，则只需再次调用sess.run(tf.initialize\_all\_variables())

weights变量一般作为trainable参数，Optimizer会自动修改Variable对象的值。若该权重不参与训练，可以设置参数trainable=False

weight\_no = tf.Variable(0, trainable=False)

在TensorFlow中，真正的循环依赖关系是无法表示的，这并非坏事。在实际使用中，完全可通过对数据流图进行有限次的复制，然后将它们并排放置，并将代表相邻迭代轮次的副本的输出与输入串接。该过程通常被称为数据流图的“展开”unrolling

保存训练检查点：周期性地保存所有变量，创建检查点checkpoint文件，并在必要时从最近的检查点恢复训练。默认情况下，Saver对象 只会保留最近的5个文件

tf.to\_float(weights)

tf.cast(weights, “float”)

tf.reshape(X, [None, 28, 28, 1])

tf.range(-1, 3)

张量逐元素运算+-\*/, //, %, \*\*, < <= > >= & | ^

tf.matmul(X, W) 矩阵乘

tf.transpose(W)

将几个行向量的特征 格式化为矩阵：行（样本），列（特征）

features = tf.transpose(tf.pack([sepal\_length, sepal\_width]))

将类名称转换为从0开始计的类别索引

label = tf.to\_int32(tf.argmax(tf.to\_int32(tf.pack([

tf.equal(label\_name, [“Iris-setosa”]),

tf.equal(label\_name, [“Iris-versicolor”]),

tf.equal(label\_name, [“Iris-virginica”]) ])), 0)

回归: 总平方差(回归损失函数)

loss = tf.reduce\_sum(tf.squared\_difference(Y\_predicted, Y))

分类问题

字符串特征转数值型特征？

为什么不能为每个可能的取值分配一个数值？如用1代表一等船票，2 and 3分别代表二，三等船票，因为这种方式会为这些取值强加一种实际并不存在的线性关系。我们不能说“三等票是一等票的3倍“

正确的做法是将每个属性特征扩展为N维的布尔型特征，每个可能的取值对应一维。若具备该属性，则相应的维度上取值为1.这样就可以使模型独立地学习到每个可能的取值的重要性。

对于只可能取两种值的属性，如果性别，用单个变量来表示已经足够，这是因为可表达这些值之间的线性关系。如令female = 1, male = 0, 则male = 1 – female, 因此单个权值便可学习同时表示两种状态

两类问题：交叉熵

tf.sigmoid(f(X, W, b) sigmoid （概率值）

loss = sum(yi\*log(y\_predictedi )+ (1-yi)log(1-y\_predictedi)) i为case

loss = tf.reduce\_mean(

tf.nn.sigmoid\_cross\_entropy\_with\_logits(Y\_predicted, Y))

注：

交叉熵与香农熵区别：

香农熵 H = -sum(yi\*log(yi ))

物理意义：已知字符串中每个字符出现的概率 yi，采用最优编码方案，则字符串编码每个字符的平均位数

交叉熵 H = -sum(yi\*log(y\_predictedi ))

实际中，对符号编码时，得到的概率是y\_predictedi 而非真实概率 yi，则每个符号所需的编码长度会更大，交叉熵允许用户以次优编码方案对字符串编码。

当建模得到的概率是y\_predictedi ＝＝真实概率 yi时，交叉熵取得最小值。交叉熵度量的是模型分布与真实分布的吻合情况。若交叉熵越接近于熵，则得到的概率是y\_predictedi 越逼近真实概率yi

多类问题：

tf.nn.softmax(f(X, W, b)) 概率值

单样本的所有类别概率和为1, 若<1, 则意味着存在一些隐藏的类别； 若>1,则说明每个样本可能同时属于多个类别

loss = -sum(sum(yic\*log(y\_predictedic ))) i为case, c为类别

若每个样本只对应单个类别

loss = tf.reduce\_mean(

tf.nn.spare\_softmax\_cross\_entropy\_with\_logits(Y\_predicted, Y))

注意：类别可以为0, 1, 2, 3, ...

若每个样本含多个类别，即类别0的概率，类别1的概率，。。。

loss = tf.reduce\_mean(

tf.nn.softmax\_cross\_entropy\_with\_logits(Y\_predicted, Y))

注意：类别只能为one-hot编码

例子：手写体

softmax regression

softmax模型可以用来给不同的对象分配概率。我们训练更加精细的模型时，最后一步也需要用softmax来分配概率。

y = softmax(xW + b)

损失函数：交叉熵cross-entropy

Hy’(y) = - sum (yi’log(yi)) 度量真实分布yi’与预测分布 yi的差异性信息

class CNN(object):

def \_\_init\_\_(self):

self.keep\_prob = 0.5

self.y\_class = 10

self.define\_graph()

self.sess = tf.Session()

self.sess.run(tf.global\_variables\_initializer())

# 保存训练检查点

self.saver = tf.train.Saver()

self.initial\_epoches = 0

ckpt = tf.train.get\_checkpoint\_state(os.path.dirname(\_\_file\_\_))

if ckpt and ckpt.model\_checkpoint\_path:

self.saver.restore(self.sess, ckpt.model\_checkpoint\_path)

self.initial\_epoches = int(ckpt.model\_checkpoint\_path.rsplit('-', 1)[1])

**def inference(self, X):**

...

return Y\_predicted

@staticmethod

**def loss(Y, Y\_predicted):**

return -tf.reduce\_sum(Y \* tf.log(Y\_predicted))

tf.argmax 给出某个tensor对象在某一维上的其数据最大值所在的索引值

@staticmethod

**def acc(Y, Y\_predicted):**

correct\_prediction = tf.equal(tf.argmax(Y\_predicted, 1), tf.argmax(Y, 1))

return tf.reduce\_mean(tf.cast(correct\_prediction, "float"))

因为TensorFlow拥有一张描述你各个计算单元的图，它可以自动地使用反向传播算法(backpropagation algorithm)来有效地确定你的变量是如何影响你想要最小化的那个成本值的。然后，TensorFlow会用你选择的优化算法来不断地修改变量以降低成本

用梯度下降算法训练你的模型，微调你的变量，不断减少成本

为什么梯度下降法需要学习率？即Wi+1 = Wi – lr\*gradient(loss(Wi))

因为梯度向量的长度实际上是一个在“损失函数单元”中而非“权值单元”中度量的量，因此需要对梯度进行缩放，使其能够与权值相加

反向传播算法：据链式法则计算梯度，每一层的导数＝后一层的导数\*前一层输出

@staticmethod

**def optimize(total\_loss):**

return tf.train.AdamOptimizer(1e-4).minimize(total\_loss)

**def define\_graph(self, ):**

self.X = tf.placeholder(tf.float32, [None, 28, 28, 1])

self.Y = tf.placeholder(tf.float32, [None, 10])

Y\_predicted = self.inference(self.X)

self.total\_loss = self.loss(self.Y, Y\_predicted)

self.optimizer = self.optimize(self.total\_loss)

self.accuracy = self.acc(self.Y, Y\_predicted)

该循环的每个步骤中，我们都会随机抓取训练数据中的128个批处理数据点，使用一小部分的随机数据来进行训练被称为随机训练（stochastic training），在理想情况下，我们希望用我们所有的数据来进行每一步的训练，因为这能给我们更好的训练结果，但显然这需要很大的计算开销。所以，每一次训练我们可以使用不同的数据子集，这样做既可以减少计算开销，又可以最大化地学习到数据集的总体特性。

**def train(self, ds\_train, batch\_size=128, epoches=5):**

total\_num = ds\_train.num\_examples

epoch = int(math.ceil(total\_num/float(batch\_size)))

for i in range(self.initial\_epoches, epoches):

print('Epoch {0}/{1}'.format(i+1, epoches))

for j in range(epoch):

**batch = ds\_train.next\_batch(batch\_size)**

\_, loss\_value, acc\_value = self.sess.run([self.optimizer, self.total\_loss, self.accuracy], feed\_dict={self.X: batch[0].reshape([-1, 28, 28, 1]), self.Y: batch[1].reshape([-1, 10])})

print('{0}/{1} [........] - loss: {2} - acc: {3}'.format(

(j+1)\*batch\_size, total\_num, loss\_value, acc\_value))

if epoches % 2 == 0:

self.saver(self.sess, 'mnist-model', global\_step=epoches)

**self.saver.save(self.sess, 'mnist-model', global\_step=epoches)**

return self

def test(self, ds\_test, batch\_size=32):

loop = int(math.ceil(ds\_test.images.shape[0] / float(batch\_size)))

accs = []

for i in range(loop):

batch = ds\_test.next\_batch(batch\_size)

accs.append(self.sess.run(self.accuracy, feed\_dict={

self.X: batch[0].reshape([-1, 28, 28, 1]),

self.Y: batch[1].reshape([-1, 10])}))

accuracy\_mean = sum(accs) / len(accs)

print('test accuracy %g' % accuracy\_mean)

多层卷积网络

权重初始化

为了创建这个模型，我们需要创建大量的权重和偏置项。这个模型中的权重在初始化时应该加入少量的噪声来打破对称性以及避免0梯度。由于我们使用的是ReLU神经元，因此比较好的做法是用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为0的问题（dead neurons）

with tf.name\_scope("conv1"):

W\_conv1 = tf.Variable(**tf.truncated\_normal**([5, 5, 1, 32], stddev=0.1))

b\_conv1 = **tf.Variable**(tf.constant(0.1, shape=[32]), name="bias")

c1 = **tf.nn.conv2d**(X, W\_conv1, strides=[1, 1, 1, 1], padding='SAME')

h\_conv1 = **tf.nn.relu**( c1 + b\_conv1)

h\_pool1 = **tf.nn.max\_pool**(h\_conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

strides参数的格式与输入向量相同，即(image\_batch\_size\_stride, image\_height\_stride, image\_width\_stride, image\_channels\_stride)

激活函数

函数单调 （便于梯度下降法寻找局部极值点成为可能）

函数可微分（从而能够计算梯度）

tf.nn.relu, tf.sigmoid, tf.tanh,

归一化层

一般对tf.nn.relu的输出进行归一化

tf.nn.local\_response\_normalization

池化层

tf.nn.max\_pool, tf.nn.avg\_pool

为了减少过拟合，常在输出层之前加入dropout

TensorFlow的tf.nn.dropout操作除了可以屏蔽神经元的输出外，还会自动处理神经元输出值的scale。所以用dropout的时候可以不用考虑scale。

keep\_prob = tf.placeholder("float")

h\_fc1\_drop = **tf.nn.dropout**(h\_fc1, keep\_prob)

输出层

y\_conv = **tf.nn.softmax**(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

获取层张量的shape

sess.run(h\_pool1.get\_shape())

高级层

加载图像

建议在训练之前对图像进行预处理并将预处理结果保存下来。每次加载图像时才对其进行处理是不推荐的做法

当输入为一幅图像时，不应使用tf.to\_float, 而应使用tf.image.convert\_image\_dtype,该方法将以恰当的方式调整各分量以表示顔色值。

预处理包括对图像裁剪，缩放，灰度调整，翻转，扭曲处理等等，以使输入给网络的训练信息多样化。虽然这个步骤会进一步增加处理时间，但却有助于缓解过拟合现象

tf.image.rag\_to\_grayscale

tf.image.convert\_image\_dtype tf.uint8 → tf.float，且归一化

tf.image.central\_crop

tf.image.crop\_to\_bounding\_box

tf.image.pad\_to\_bounding\_box

tf.image.resize\_image\_with\_crop\_or\_pad

tf.image.flip\_left\_right

tf.image.flip\_up\_down

tf.image.adjust\_brightness

tf.image.adjust\_contrast

tf.image.adjust\_hue

tf.image.adjust\_saturation

tf.image.rgb\_to\_hsv

含权值初始化，偏置初始化，激活函数

tf.contrib.layers.convolution2d

含权值初始化，偏置初始化

tf.contrib.layers.fully\_connected

深度值的增加，减少使用该网络所需的计算量

from **tf.contrib.layers** import convolution2d, max\_pool2d, fully\_connected

def inference(self, X):

with tf.name\_scope("conv1"):

conv1 = convolution2d(X, 32, (5, 5))

pool1 = max\_pool2d(conv1, (2, 2))

with tf.name\_scope("conv2"):

conv2 = convolution2d(pool1, 64, (5, 5))

pool2 = max\_pool2d(conv2, (2, 2))

with tf.name\_scope("fc1"):

pool2\_flat = tf.reshape(pool2, [-1, 7 \* 7 \* 64])

fc1 = fully\_connected(pool2\_flat, 1024)

with tf.name\_scope("fc2"):

fc1\_drop = tf.nn.dropout(fc1, self.keep\_prob)

y\_conv =fully\_connected(fc1\_drop, self.y\_class, activation\_fn=tf.nn.softmax)

return y\_conv

官方站点：

tensorflow.org/get\_started/

import tensorflow as tf

import numpy as np

features = [tf.contrib.layers.real\_valued\_column("x", dimension=1)]

estimator = tf.contrib.learn.LinearRegressor(feature\_columns=features)

x\_train = np.array([1., 2., 3., 4.])

y\_train = np.array([0., -1., -2., -3.])

x\_eval = np.array([2., 5., 8., 1.])

y\_eval = np.array([-1.01, -4.1, -7, 0.])

input\_fn = tf.contrib.learn.io.numpy\_input\_fn({"x":x\_train}, y\_train, batch\_size=4, num\_epochs=1000)

eval\_input\_fn = tf.contrib.learn.io.numpy\_input\_fn({"x":x\_eval}, y\_eval, batch\_size=4, num\_epochs=1000)

estimator.fit(input\_fn=input\_fn, steps=1000)

print("train loss: %r"% estimator.evaluate(input\_fn=input\_fn))

print("eval loss: %r"% estimator.evaluate(input\_fn=eval\_input\_fn))

Pycharm tensorflow ImportError but works fine with Terminal?

open PyCharm from the command line and everything works now

# 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

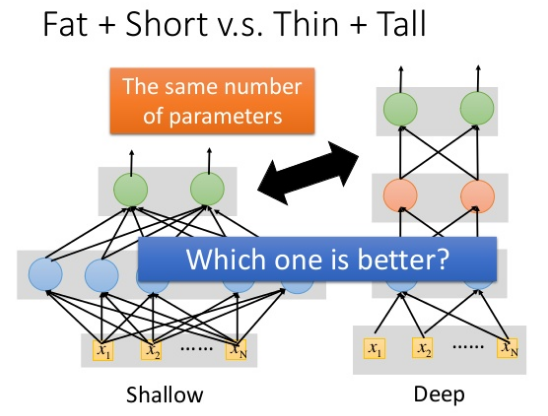
Training set

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

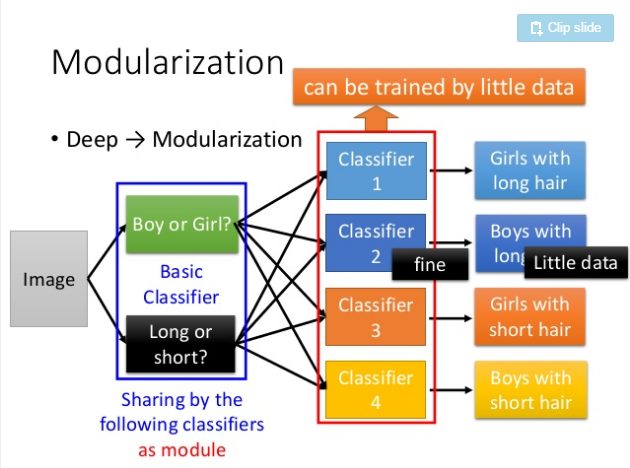
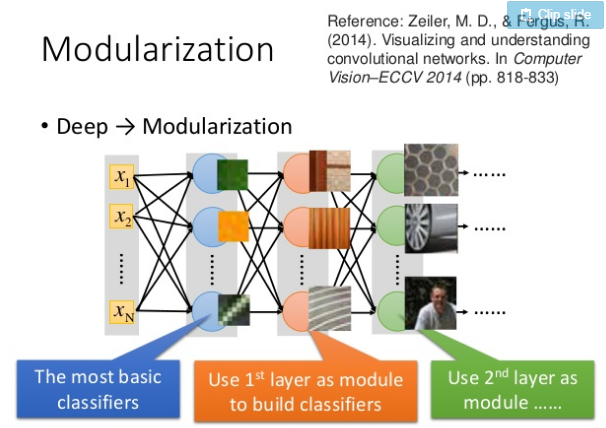


A two layers of logic gates can represent **any Boolean function**

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

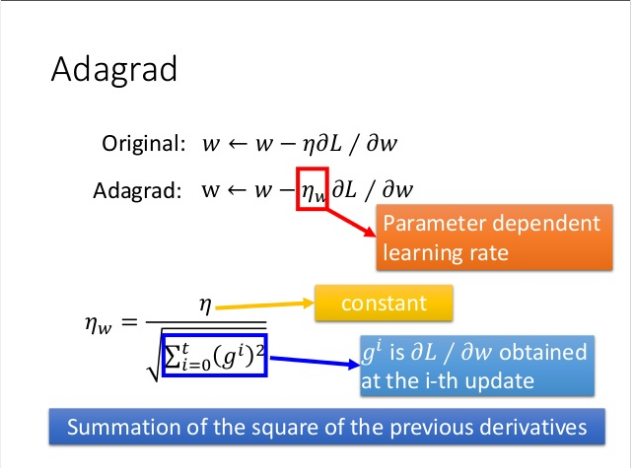
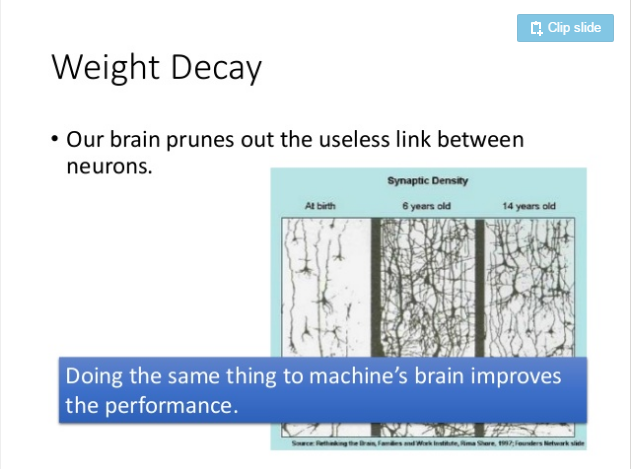
Using multiple layers of neurons to represent some functions are much simpler -> less parameters, less data?

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

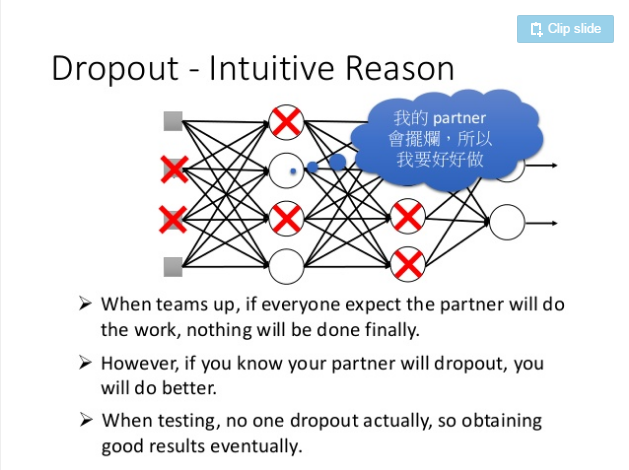
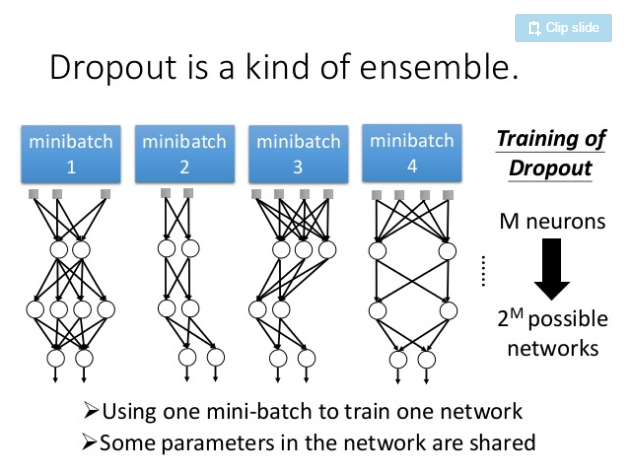
小梯度，平坦，则大步伐；大梯度，陡峭，则小步伐

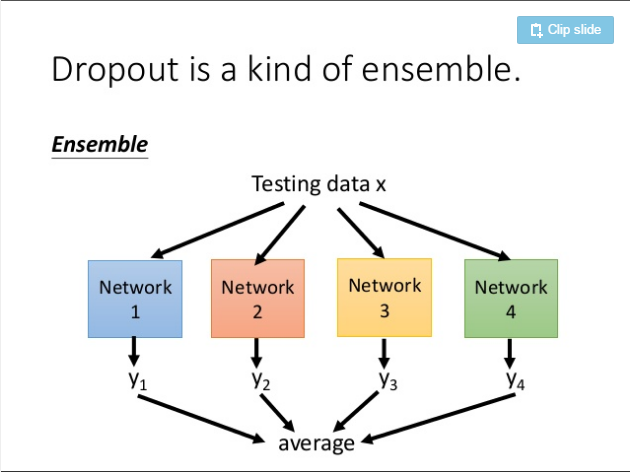
 

Weight Decay

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：

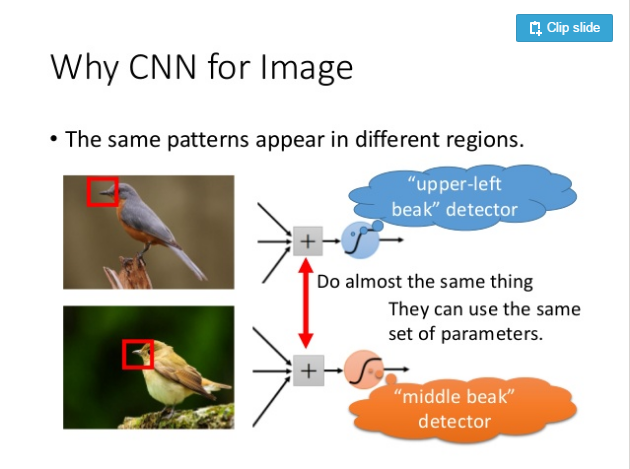
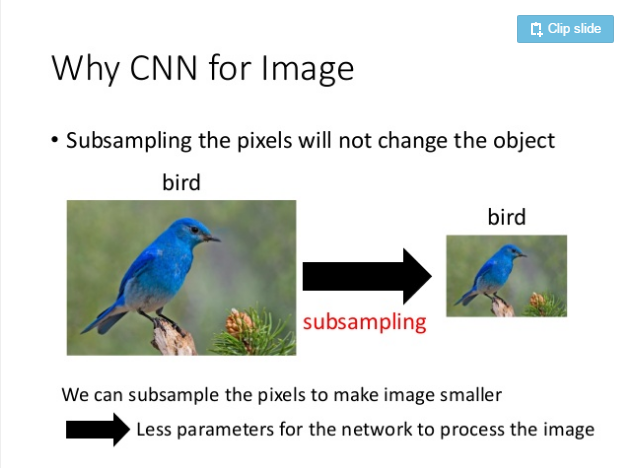
 

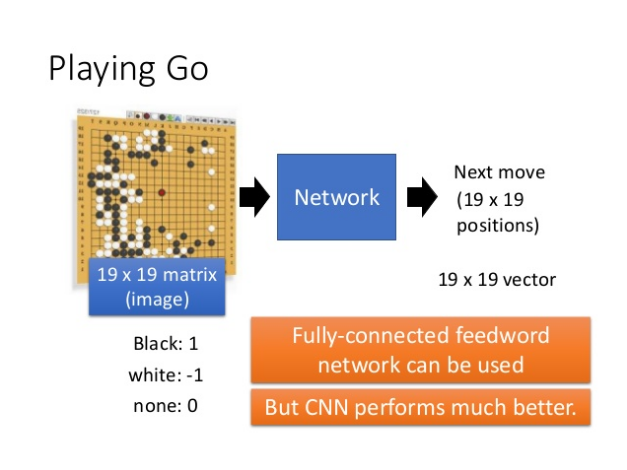
### CNN

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)

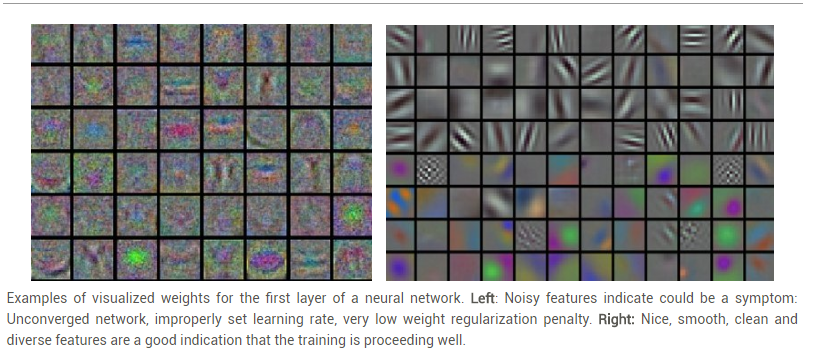
网络权值的解释？W = 模板

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

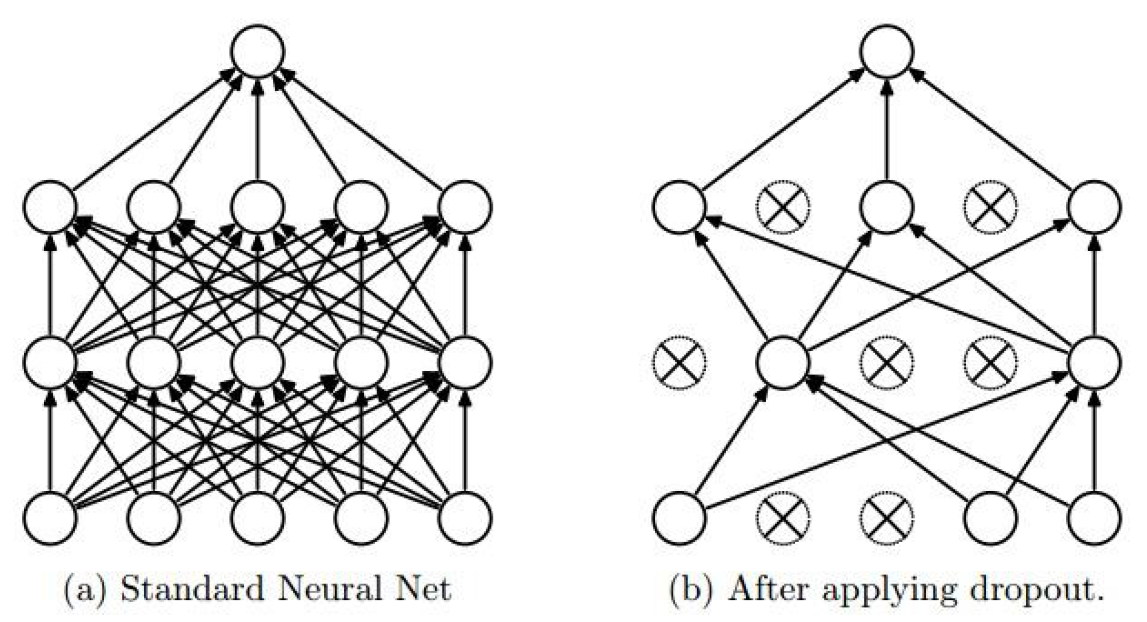
Mini-batch SGD

Loop:

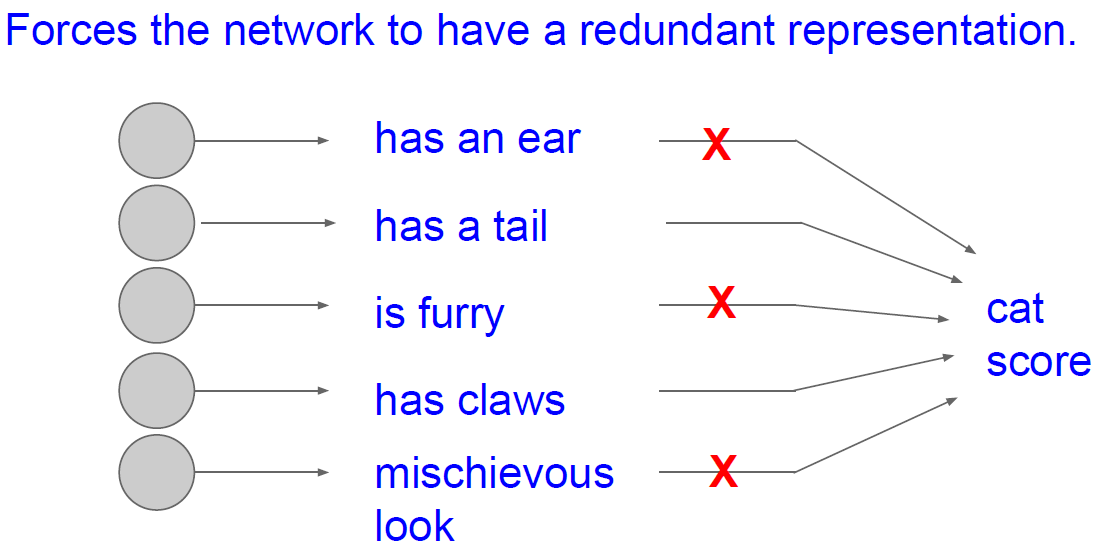
1. Sample a batch of data
2. Forward prop it through the graph, get loss
3. Backprop to calculate the gradients
4. Update the parameters using the gradient

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.

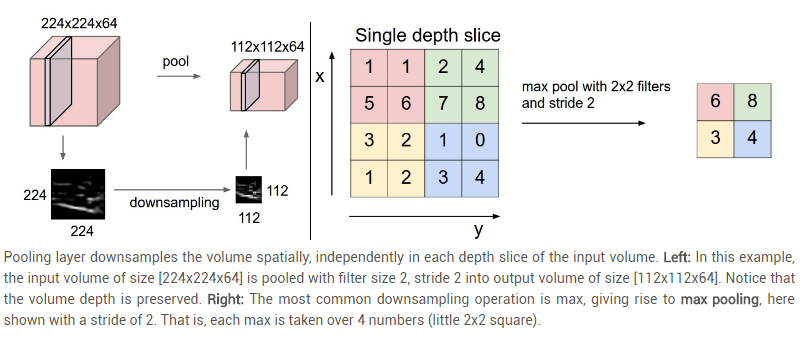
假设不一定对：Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than 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

知识库：关于世界的知识用形式化的语言进行硬编码，计算机可以使用逻辑推理规则来自动地理解这些形式化语言中的声明。

机器学习：AI系统需要具备自己获取知识的能力，即从原始数据中提取模式的能力。

简单的机器学习算法的性能在很大程度上依赖于给定数据的表示。表示的选择会对机器学习算法的性能产生巨大的影响。

举例：MRI扫描的单一像素与分娩过程中并发症之间的相关性微乎其微

传统方式：基于领域知识的特征工程

表示学习：使用机器学习来发掘表示本身。学习到的表示往往比手动设计的表示表现得更好，并且它们只需最少的人工干预，就能让AI系统迅速适应新的任务。

深度学习：通过其他较简单的表示来表达复杂表示，解决了表示学习中的核心问题。（通过较简单的概念构建复杂的概念）。将大千世界表示为嵌套的层次概念体系（由较简单概念间的联系定义复杂概念、从一般抽象概括到高级抽象表示）

联结主义的中心思想：当网络将大量简单的计算单元连接在一起时可以实现智能行为。单独神经元或小集合的神经元不是特别有用。

分布式表示，反向传播算法

一个粗略的经验法则：监督深度学习算法在每类给定约5000个标注样本情况下一般将达到可以接受的性能，当至少有1000万个标注样本的数据集用于训练时，它将达到或超过人类表现。

应用：语音识别，行人检测，图像分割，交通标志分类

### 矩阵论

衡量向量大小：

L0范数：向量中非零元素的个数

L1范数：向量中元素的绝对值之和

L2范数：欧几里得范数

最大范数：向量中元素的最大绝对值

衡量矩阵大小：

Frobenius范数：欧几里得范数

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**

逆矩阵主要是作为理论工具使用的，并不会在大多数软件应用程序中实际使用。这是因为逆矩阵在数字计算机上只能表现出有限的精度.常用奇异值分解

每个实对称矩阵都可以分解成实特征向量和实特征值，实特征矩阵的特征分解也可以用于优化二次方程f(x) = xTAx, s.t. to ||x|| = 1. 当x等于A的某个特征向量时，f将返回对应的特征值。在限制条件下，函数f的最大值是最大特征值，最小值是最小特征值。

所有特征值都是非负数的矩阵，称为半正定的positive semi-definite. 半正定矩阵受到关注是因为它们保证任意x, xTAx >= 0

奇异值分解：每个实数矩阵都有一个奇异值分解

A = U D VT

假定A是m\*n矩阵，

则U是m\*m正交矩阵， U的列向量称为左奇异向量，是AAT的特征向量

V是n\*n正交矩阵，V的列向量称为右奇异向量，是ATA的特征向量

D是m\*n奇异值对角阵。非零奇异值是AAT也是ATA的特征值平方根

迹运算：Tr(A) = sum(aii) Tr(AAT) = A矩阵范数的平方

行列式：将方阵A映射到实数的函数。行列式的绝对值可以用来衡量矩阵参与矩阵乘法后空间扩大或者缩小了多少。如果行列式是0，那么空间至少沿着某一维完全收缩了，使其失去了所有的体积；

### 概率与信息论

概率论：不确定声明以及在不确定性存在的情况下进行推理

信息论：量化概率分布中的不确定性总量

在很多情况下，使用一些简单而不确定的规则要比复杂而确定的规则更为实用

一个随机变量只是对可能的状态的描述； 它必须伴随着一个概率分布来指定每个状态的可能性。

离散型变量和概率质量函数

连续型变量和概率密度函数

联合概率分布，边缘概率分布，条件概率

期望：函数f(x)关于某分布P(x)的期望是指，当x由分布P(x)产生，f作用于x时，f(x)的平均值。对于离散型随机变量求和，对于连续型随机变量求积分

方差：依据概率分布进行采样时，随机变量x的函数值f(x)会显现多大的差异

协方差：两个变量线性相关性的强度，若协方差是正的，那么两个变量都倾向于同时取得相对较大的值。如果协方差是负的，那么其中一个变量倾向于取得相对较大的值的同时，另一个变量倾向于取得相对较小的值。

相关系数：归一化的协方差

协方差，相关性，独立性区别？

如果两个变量相互独立，那么它们的协方差为0；

如果两个变量的协方差不为0,那么它们一定是相关的；

如果两个变量的协方差等于0,那么它们之间一定没有线性关系 ，可能还有非线性关系

独立性比0协方差的要求更强，因为独立性不仅没有线性关系，还排除了非线性的关系。

协方差矩阵：元素是两个变量的协方差

分布

Bernoulli分布：二值随机变量分布

Multinoulli分布：k个不同状态的随机变量分布

高斯分布：正态分布

当我们由于缺乏关于某个实数上分布的先验知识而不知道该选择怎样的形式时，正态分布是默认的比较好的选择。两个原因：

1. 建模的很多分布的真实情况是比较接近正态分布的。 由中心极限定理表明很多独立随机变量的和近似正态分布。
2. 正态分布是对模型加入 的先验知识量最少的分布。

指数分布

Laplace分布

Dirac分布

经验分布：Dirac分布的组合

经验分布将概率密度1/m赋给m个点中的每一个，这些点是给定的数据集。当我们在训练集上训练模型时，可以认为从这个训练集上得到的经验分布指明了采样来源的分布。

另一种观点，经验分布是训练数据的似然最大的那个概率密度函数

混合分布，高斯混合模型

高斯混合模型是概率密度的万能近似器，即任何平滑的概率密度都可以用具有足够多组件的高斯混合模型以任意精度来逼近。

先验概率：表明在测测到x之前传递给模型关于类别的信念

后验概率：在观测到x之后进行计算

函数：

Logistic sigmoid函数

Softplus函数

信息论的基本想法：一个不太可能的事件居然发生了，要比一个非常可能的事件发生，能提供更多的信息。通过这种基本想法来量化信息：即非常可能发生的事件信息量要比较少； 较不可能发生的事件具有更高的信息量

自信息：I(x) = -log P(x)

香农熵：对整个概率分布中的不确定性总量进行量化

H(x) = sum( - P(x) log P(x) )

那些接近确定性的分布（输出几乎可以确定）具有较低的熵； 那些接近均匀分布的概率分布具有较高的熵。

KL散度：两个概率分布的差异

DKL(P||Q) = sum( - P(x) log P(x)/Q(x) )

KL散度是非负的，并且衡量的是两个分布之间的差异，它经常被用作分布之间的某种距离。然而，它并不是真的距离，因为它不是对称的，即DKL(P||Q) != DKL(Q || P)

结构化概率模型

机器学习的算法经常会涉及在非常多的随机变量上的概率分布。通常，这些概率分布涉及的直接相互作用都是介于 非常少的变量之间的。使用单个函数来描述整个联合概率分布是非常低效的，我们可以把概率分布分解成许多因子的乘积形式，而不是使用单一的函数来表示概率分布。

当用图来表示这种概率分布的分解时，我们把它称为结构化概率模型或者图模型

这些图模型表示的分解仅仅是描述概率分布的一种语言

### 数值计算

机器学习算法通常通过迭代过程更新解的估计值来解决数学问题的算法，而不是通过解析过程推导出公式来提供正确解的方法，常见的操作包括优化（找到最小化或最大化函数值的参数）和线性方程组的求解

条件数：函数相对于输入的微小变化而变化的快慢程度

考虑函数f(x) = A-1x, 条件数为矩阵A的最大最小特征值的模之比

优化：改变x以最小化或最大化某个函数f(x)的任务

一阶优化算法：基于梯度的优化方法，多维情况下，临界点是梯度中所有元素都为零的点。

二阶优化算法：使用Hessian矩阵的优化算法，如牛顿法

Hessian矩阵：输入多维，输出一维，输出对输入的二阶导，当Hessian矩阵的条件数很差时，梯度下降法会表现得很差，因为一个方向上的导数增加很快，另一个方向增加得很慢； 同时病态条件会导致很难选择合适的步长，步长必须足够小，以免冲过最小而向具有较强正曲率的方向上升，这通常意味着步长太小，以至于在其他较小曲率的方向上进展不明显。

梯度为0，称为临界点，可能 是局部极大点、全局极小点或是鞍点。需要二阶导确定。

当Hessian是正定（所有特征值是正的），则该临界点是全局极小点；

当Hessian是负定，则为全局极大点。

当Hessian特征值中至少一个是正，一个是负，则x是f某个横截面的局部极大点，另一个横截面的局部极小点。

仅当附近的临界点是最小点（即Hessian是正定），牛顿法才适用，而梯度下降不会被吸引到鞍点。

优化算法没有理论保证，因为在深度学习中使用的函数相当复杂，所以深度学习算法往往缺乏理论保证。

凸优化算法只对凸函数适用，即Hessian处处半正定的函数。

约束优化：在x的某些集合S中找f(x)的最大值或最小值。

KKT方法 == 广义Lagrange函数（允许不等式约束）， Lagrange乘子法的推广（仅允许等式约束）

### 机器学习基础

机器学习本质上属于应用统计学，更多地关注于如何用计算机统计地估计复杂函数

输入缺失分类？ Goodfellow的深度概率模型

在医疗诊断中经常出现，因为很多类型的医学测试是昂贵的，对身体有害的。

当一些输入可能丢失时，学习算法必须学习一组函数，而不是单个分类函数。每个函数对应着分类具有不同缺失输入子集的x.

机器学习的任务: 分类，回归，机器翻译，结构化输出（含图像分割），异常检测，缺失值填补，去噪、密度估计（学习样本采样空间的概率密度函数或概率质量函数）

性能度量： accuracy, error rate, …

经验: 数据集

无监督学习试图显式或隐式地学习出概率分布或该分布一些有意思的性质。

无监督学习和监督学习互相转换？

通过概率的链式法则，无监督学习p(x) = p(x1) p(x2|x1)…等效为n个监督学习问题。

通过联合分布，监督学习p(y|x) = p(x, y) / p(x), 转化为无监督学习

容量、过拟合和欠拟合

当我们只能观测到训练集时，如何才能影响测试集的性能呢？统计学习理论

假定训练集和测试集中的样本是独立同分布的，也就是由共享的潜在的数据生成分布采样。那么训练误差和测试误差之间的直接联系是：随机模型训练误差的期望和该模型测试误差的期望是一样的。

通过调整模型的容量，可以控制模型是否偏向于过拟合或欠拟合。通俗地讲，模型的容量是指其拟合各种函数的能力。容量低的模型可能很难拟合训练集，容量高的模型可能会过拟合，因为记住了不适用于测试集的训练集性质。

VC维： 度量二元分类器的容量，定义为该分类器能够分类的训练样本的最大数目

统计学习理论结论：训练误差和泛化误差之间差异的上界随着模型容量增长而增长，随着训练样本增多而下降。

泛化误差是一个关于模型容量的U形曲线函数

没有免费午餐定理：在所有可能的数据生成分布上平均之后，每一个分类算法在未事先观测的点上都有相同的错误率。这个结论仅在我们考虑所有可能的数据生成分布时才成立。机器学习研究的目标不是找一个通用学习算法或是绝对最好的学习算法，而是理解什么样的分布与人工智能获取经验的“真实世界”相关，以及什么样的学习算法在我们关注的数据生成分布上效果最好。

正则化：指修改学习算法，使其降低泛化误差而非训练误差

为什么超参数不能在训练集上学习？

如果在训练集上学习超参数，这些超参数总是趋向于最大可能的模型容量，导致过拟合。

训练集(训练集+验证集) + 测试集

一个小规模的测试集意味着平均测试误差估计的统计不确定性，使得很难判断算法A是否比算法B在给定的任务上做得更好

K折交叉验证：测试误差可以估计为k次计算后的平均测试误差

频率派视角：

真实参数theta是未知的定值，而点估计theta是考虑数据集上函数的随机变量

点估计， 函数估计（函数空间中的一个点估计）

估计的偏差： 偏差真实函数或参数的误差期望

估计的方差： 数据上任意采样可能导致的估计期望的偏差

均值的标准差在机器学习实验中非常有用，因为我们通常用测试集样本的误差均值来估计泛化误差。测试集中样本的数量决定了这个估计的精确度。中心极限定理告诉我们均值会接近一个高斯分布，我们可以用标准差计算出真实期望落在选定区间的概率。

以均值u为中心的95%置信区间是：(miu - 95%phi, miu + 95%phi)是基于均值miu,标准差phi的高斯分布。在机器学习实验中，我们通常说算法A比算法B好，是指算法A的误差的 95%置信区间的上界小于算法B的误差的95%置信区间的下界

贝叶斯统计视角：

用概率反映知识状态的确定性程度，数据集能够被直接观测到，因此不是随机的，另一方面真实参数theta是未知或不确定的，因此可以表示为随机变量

由贝叶斯规则： p(theta | X) = p(X |theta) p(theta) / p(X)

在贝叶斯估计常用的情景下，先验开始是相对均匀的分布或高熵的高斯分布，观测数据通常会使后验的熵下降，并集中在参数的几个可能性很高的值。

贝叶斯估计与最大似然估计区别？

1. 不像最大似然方法预测时使用theta的点估计，贝叶斯方法使theta的全分布
2. 贝叶斯先验分布。先验能够影响概率质量密度朝参数空间中偏好的先验的区域偏移，实验中先验通常表现为偏好更简单或更光滑的模型

当训练数据有限时，贝叶斯方法通常泛化更好，当训练样本数目很大时，通常会有很大的计算代价。

结合二者优点的最大后验点估计MAP (max a posteriori)

选择后验概率最大的点

theta = argmax p(theta | X) = argmax log p(X | theta) + log p(theta)

第一项对应标准的对数似然项；第二项对应着先验分布

MAP估计的优势是能够利用来自先验的信息，这些信息无法从训练数据中获得。该附加信息有助于减少最大后验点估计的方差，但代价是增加了偏差。

许多带正则化的估计方法，如正则化的最大似然学习，可以被解释为贝叶斯推断的MAP推断, log p(theta) 等价于正则化

SVM核技巧强大的两个原因：

1. 能够使用保证有效收敛的凸优化技术来学习非线性模型（关于x的函数）
2. 核函数k的实现方法通常比直接构建phi(x)再算点积高效很多

核函数为高斯核的SVM，等效于模板匹配template matching?

训练标签y相关的训练样本x变成了类别y的模板，当测试点x’到x的欧几里得距离很小，对应的高斯核响应很大时，表明x’和模板x非常相似。该模型进而会赋予相对应的训练标签y较大的权重。预测将会组合很多这种通过训练样本相似度加权的训练标签

KNN

在分类 情况中，我们可以关于one-hot编码向量c求平均，其中cy = 1, 其他的i值取ci = 0，然后我们可以解释这些one-hot编码的均值为类别的概率分布

KNN的高容量使其在训练样本数目大时能够获取较高的精度，然而计算成本很高；在训练集较小时泛化能力较差

KNN的一个弱点是它不能学习出哪一个特征比其他更具识别力

决策树

使用坐标轴相关的拆分，并且每个子节点关联到常数输出，因此有时解决不是坐标轴对齐的分类问题很费力，如二分类问题x2 > x1时分为正类

无监督学习任务：找到数据的最佳表示。

最常见的有：低维表示、稀疏表示和独立表示

PCA:

去除变量间的线性相关 （是学习表示中元素统计独立标准的第一步，要实现完全 独立性，表示学习算法还必须去掉变量间的非线性关系）

学习数据的正交线性变换，通过输入空间的一个旋转（由W确定，数据的右奇异向量）

线性投影，使最大方差的方向和新空间的轴对齐

聚类 问题本身是病态的，因为没有单一的标准去度量聚类的数据在真实世界中效果如何。我们不知道聚类的性质（类中元素到类中心距离）是否很好地对应到真实世界的性质，可能有许多不同的聚类对应到现实世界的某些属性，如红色卡车，红色汽车，灰色卡车，灰色汽车，运行两个聚类：可能得汽车和卡车各聚一类，或红色和灰色各聚一类

随机梯度下降的核心：梯度是期望，期望可以可使用小规模的样本近似估计。核方法需要构建m\*m的核函数k(xi, xj), 若数据集是几十亿样本时，该计算量不可接受，然而mini-batch随机梯度下降，每一步随机梯度下降更新的计算量不取决于熟练训练集的大小m

促使深度学习发展的挑战

为何处理高维数据时在新样本上泛化特别困难？以及为何在传统机器学习中实现泛化的机制不适合学习高维空间中复杂的函数？

1. 维数灾难

估计某点x处的概率密度 = x处理单位体积内训练样本的数目除以训练样本的总数

低维时，新数据点附近都有训练点；若高维时，新数据点附近没有训练点，如何判别？而许多传统机器学习算法只是简单地假设在一个新点的输出应大致和最接近的训练点的输出相同。也就是说，为了更好地泛化，传统机器学习算法需要假设“平滑性”和“局部不变性”，即学习在函数不应在小区域内发生很大的变化。许多简单算法完全依赖于此先验达到良好的泛化，但是无法推广去解决人工智能级别任务中的统计挑战如语音识别，对象识别等

举例：

KNN复制附近训练样本的输出

核机器也是在和附近训练样本相关的训练集输出上插值，如局部核，核函数k(u, v)在u = v 时很大，当u和v距离拉大时而减小。局部核可以看作执行模板匹配的相似函数，用于度量测试样本x和每个训练样本x有多么相似

只要在要学习的真实函数的峰值和谷值处有足够多的样本，那么平滑性假设和相关的无参数学习算法的效果都非常好。

能否有效地表示复杂的函数以及所估计的函数是否可以很好地泛化到新的输入？

答案：通过额外假设生成数据的分布来建立区域间的依赖关系

如训练样本数远小于棋盘上的黑白方块数，假设目标函数是周期性的，就容易解决棋盘问题。

深度学习基于更通用的假设：假设数据由特征组合产生，这些特征可能来自一个层次结构的多个层级。深度的分布式表示带来的指数增益有效地解决了维数灾难带来的挑战

1. 流行

流形：指连接在一起的区域，数学上是指一组点，且每个点都有其领域。给定任意点，其流形局部看起来像是欧几里得空间。

如地球视为二维平面，实际上它是三维空间中的球状流形。

若机器学习算法学习整个Rn上有趣变化的函数，那么很多机器学习问题看上去都是无望的。（得需要多少的训练集？）流形假设：Rn中大部分区域都是无效的输入，有意义的输入只分布在包含少量数据点的子集构成的一组流形中。学习函数的输出中，有意义的变化都沿着流形的方向或仅发生在我们切换到另一流形中。

在人工智能的一些场景中，如现实生活中的图像、声音或文本的概率分布都是高度集中的，流形假设是近似对的。

集中的概率分布不足以说明数据位于一个相当小的流形中，还必须确保，所遇到的样本和其他样本相互连接，每个样本被其他高度相似 的样本包围，而这些高度相似的样本可以通过变换来遍历该流形得到。想像一下图片空间的流形：逐渐变暗或变亮、逐步移动或旋转图中对象、逐渐改变对象表面的颜色等。

大多数应用中很有可能会涉及到多个流形，如人脸图像的流形不太可能连接到猫脸图像的流形。

当数据位于低维流形中时，使用流形中的坐标而非Rn中的坐标表示机器学习数据更为自然。如道路是嵌入在三维空间的一维流形，我们用一维道路中的地址号码确定地址，而非三维空间中的坐标。

神经网络性能的大部分改进可归因于两个因素：

* 较大的数据集减少了统计泛化对神经网络的挑战的程度
* 神经网络由于更强大的计算机和更好的软件基础设施已经变得更大

小部分的改进：算法上的变化

* 如损失函数的交叉熵族替代均方误差，使用交叉熵损失大提高了具有sigmoid and softmax输出的模型的性能，而当使用均方误差损失时会存在饱和和学习缓慢的问题。
* 分段线性隐藏单元来替代sigmoid隐藏单元，在深度整流网络中的学习比在激活函数具有曲率或两侧饱和的深度网络中的学习更容易

### 深度前馈网络

为了实现统计泛化而设计出的函数近似机

大多数神经网络通过仿射变换之后紧跟着一个被称为激活函数的固定非线性函数来实现这个目标

循环神经网络

训练数据为我们提供了在不同训练点上取值的、含有噪声的f\*(x)的近似实例

层可以想像成向量到向量的单个函数，或由许多并行操作的单元组成

线性模型有明显的缺陷，那就是该模型的能力被局限在线性函数里，所以它无法理解任何两个输入变量间的相互作用。

为了扩展线性模型，线性模型不直接用于x本身，而是用于一个非线性变换phi(x)上。使用phi的一个非线性变换，可以认为phi提供了一组描述x的特征，或者认为它提供了x的一个新的表示

如何选择非线性变换phi(x)?

基于领域知识的特征工程

核方法如基于RBF核的核机器上

特征表示学习 如深度学习

模型：y = f(x; theta, w) = phi(x; theta) w

人类专家可以将他们的知识编码进网络来帮助泛化，他们只需要设计那些他们期望能够表现优异的函数族phi(x; theta)即可，这种方法的优点是人类设计者只需要寻找正确的函数族即可，而不需要去寻找精确的函数

损失函数

用于非凸损失函数的随机梯度下降没有收敛性保证，并且对参数的初始值很敏感。对于前馈神经网络，将所有权重值初始化为小随机数是很重要的

为什么交叉熵代价函数比均方误差或者平均绝对误差更受欢迎？

均方误差和平均绝对误差在使用基于梯度的优化方法时往往成效不佳。一些饱和的输出单元当结合这些代价函数时会产生非常小的梯度。

负的对数似然可以避免隐藏单元或输出单元输出的激活函数饱和，比如很多输出单元都会包含一个指数函数。

在大多数情况下，参数模型定义了一个分布p(y|x; theta)，并且简单地使用最大似然原理，这意味着代价函数就是负的对数似然，与训练数据和模型预测间的交叉熵等价。

输出单元

选择如何表示输出决定了交叉熵函数的形式

用于高斯输出分布的线性单元： p( y | x ) = N (y; y’, I)

最大化其对数似然此时等价于最小化均方误差

用于Bernoulli输出分布的sigmoid单元

Bernoulli分布仅需单个参数来定义，神经网络只需要预测p(y=1 | x) 即可。

代价函数-log P(y | x), 代价函数中的log抵消了sigmoid中的exp. 如果没有这个效果，sigmoid的饱和性会阻止基于梯度的学习做出好的改进。

最大似然函数几乎总是训练sigmoid输出单元的优选方法

用于multinoulli输出分布的softmax单元

softmax函数最常用作分类器的输出，来表示n个不同类上的概率分布。

输出线性层预测了未归一化的对数概率： z = WTh + b

softmax(zi) = exp(zi) / sum(exp(z))

负对数似然函数 - log softmax(zi) = - zi + log sum(exp(z))

解释：负对数似然代价函数总是强烈地惩罚最活跃的不正确预测。如果正确答案已经具有了softmax的最大输入，那么- zi项和log sum(exp(z)) = max zj = zi项将大致抵消。这个样本对于整体训练代价贡献很小，这个代价主要由其他未被正确分类的样本产生。

对数似然之外的许多目标函数对softmax函数不起作用。具体来说，那些不使用对数来抵消softmax中的指数的目标函数，当指数函数的变量取非常小的负值时会造成梯度消失，从而无法学习。特别是平方误差，对于softmax单元来说，它是一个很差的损失函数

像sigmoid一样，softmax激活函数可能会饱和，sigmoid函数具有单个输出，当它的输入极端负或者极端正时会饱和。对于softmax的情况，它有多个输出值，当输入值之间的差异变得极端时，这些输出值可能饱和。

Softmax函数提供了argmax的软化版本

乘法，加法和对数运算的梯度表现良好，除法函数在0附近会变得任意陡峭，导致任意大的梯度通常导致不稳定。

隐藏单元

ReLU: g(z) = max{0, z}

* 分段线性函数，几乎是线性的，因此保留了许多使得线性模型易于使用基于梯度的方法进行优化的属性，计算机科学的一个公共原则是，可以从最小的组件构建复杂的系统，我们可能从整流线性函数构建一个万能函数近似器
* ReLU在z = 0 处不可微，这似乎使得g对于基于梯度的学习算法无效。在实践中，梯度下降对这些机器学习模型仍然表现得足够好。部分原因是神经网络训练算法通常不会达到代价函数的局部最小值，而是仅仅显著地减小它的值
* ReLU易于优化，因为这们和线性单元非常类似
* ReLU通常作用于仿射变换之上： h = g(WT X + b)
* 初始化仿射变换的参数时，可以将b的所有元素设置成一个小的正值，例如0.1。 这使得ReLU很可能初始时就对训练集中的大多数输入呈现激活状态，并且允许导数通过
* ReLU和它的扩展都是基于一个原则，那就是如果它们的行为更接近线性，那么模型更容易优化

为什么不鼓励把sigmoid and tanh作为前馈网络中的隐藏单元？

Sigmoid单元在其大部分定义域内都饱和，当z取绝对值很大的正值时，它们饱和到一个高值，当z取绝对值很大的负值时，它们饱和到一个低值，并且仅仅当z接近0时，它们才对输入强烈敏感。Sigmoid单元的广泛饱和性会使得基于梯度的学习变得非常困难，因为这个原因，现在不鼓励将它们用作前馈网络中的隐藏单元。当使用一个合适的代价函数来抵消sigmoid的饱和性时，它们作为输出单元可以与基于梯度的学习相兼容

架构设计

更深层的网络通常能够对每一层使用更少的单元数和更少的参数，并且经常容易泛化到测试集，但是通常也更难以优化

万能近似定理universal approximation theorem: MLP能够表示任何 有限维空间到另一个有限维空间的映射（简单讲,MLP是非线性函数近似器），MLP的导数表示映射的导数

即使MLP能够表示任意函数，学习也可能因两个不同的原因而失败？

* 用于训练的优化算法可能找不到用于期望函数的参数值
* 训练算法可能由于过拟合而选择了错误的函数

具有单层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法正确地学习和泛化。在很大情况下，使用更深的模型能够减少表示期望函数所需的单元的数量，并且可以减少泛化误差

选择深度模型默许一个非常普遍的信念，那就是我们想要觉得的函数应该涉及几个更加简单的函数的组合。使用深层模型表达出了对模型可以实习的函数空间的有用偏好。具体来说，它表达了一种信念，即该函数应该由许多更简单的函数复合在一起而得到。这可能导致学习由更简单的表示所组成的表示（例如，由边所定义的角）或学习具有顺序依赖步骤的程序（例如，首先定位一组对象，然后分割它们，之后识别它们）

许多架构构建一个主链，但随后又添加了额外的架构特性，例如从层i到>i+1的跳跃连接，这些跳跃连接使得梯度更容易从输出层流向更接近输入的层

反向传播：仅指用于计算梯度的方法，不仅仅适用于多层神经网络，也可以计算任何函数的导数

随机梯度下降：基于梯度来进行学习

反向传播算法被设计为减少公共子表达式的数量而不考虑存储的开销，因此避免了重复表达式的指数爆炸。然后，其他算法可能通过对计算图进行简化来避免更多的子表达式，或者也可能通过重新计算而不是存储这些子表达式来节省内存。

反向传播算法避免了重复计算许多公共子表达式，这种表填充策略有时被称为动态规划

深度学习软件库的用户能够对使用诸如矩阵乘法、指数运算、对数运算等常用操作构建的图进行反向传播。

如矩阵乘法C = A B , 假设标量z关于C的梯度是G. 则关于A的梯度是GBT; 则关于B的梯度是ATG

大多数神经网络的代价函数大致是链式结构的，使得反向传播只有O(n)的成本

对于MLP， 计算成本主要来源于矩阵乘法，在前向传播阶段，我们乘以每个权重矩阵；在反向传播阶段，我们乘以每个权重矩阵的转置

反向传播保证梯度计算的计算数目和前向计算的计算数目是同一个量级

### 深度学习中的正则化

对学习算法的修改，旨在减少泛化误差而不是训练误差

正则化策略：模型添加约束和惩罚

* 约束和惩罚被设计为编码特定类型的先验知识
* 被设计成偏好简单模型，以便提高泛化能力
* 集成的方法，结合多个假说来解释训练数据

正则化一般是以偏差的增加换取方差的减少。一个有效的正则化是有利的“交易”，也就是能显著减少方差而不过度增加偏差

控制模型的复杂度不是找到合适规模的模型（带有正确的参数个数），最好的拟合模型（从最小化泛化误差的意义上）是一个适当正则化的大型模型

机器学习中许多线性模型，包括线性回归和PCA，都依赖于对矩阵XTX求逆。当数据生成分布在一些方向上确实没有差异时，或因为例子较少（即相对输入特征的维数来说）而在一些方向上没有观察到方差时，这个矩阵就是奇异的。在这种情况下，正则化的许多形式对应求逆XTX + alpha I， 该正则化矩阵可以保证可逆

大多数形式的正则化能够保证应用于欠定问题的迭代方法收敛

求解欠定线性方程，伪逆(XTX + alpha I)-1XT,使用正则化稳定欠定问题

参数范数惩罚

在神经网络中，参数包括每一层仿射变换的权重和偏置，我们通常只对权重做惩罚而不对偏置做正则惩罚。正则化偏置参数可能会导致明显的欠拟合

为了减少搜索空间，我们会在所有层使用相同的权重衰减

正则化系数较大，将得到一个较小的约束区域；较小的正则化系数，将得到一个较大的约束区域

L2参数正则化 = 权重衰减，岭回归，Tikhonov正则

目标：使权重更加接近原点

L1正则化

产生更稀疏的解，即最优值中的一些参数为0。L1通常足够大的正则化系数实现稀疏

L1正则化导出的稀疏性质已经被广泛用于特征选择，如LASSO模型将L1惩罚和线性模型结构，并使用最小二乘法代价函数，L1惩罚使部分子集的权重为0, 表明相应的特征可以被安全地忽略

数据集增强

即使模型已使用卷积和池化技术对部分平移保持不变，沿训练图像每个方向平移几个像素的操作通常可以大大改善泛化。还有旋转图像，缩放图像…

神经网络被证明对噪声不是非常健壮的，改善神经网络健壮性的方法之一是：

随机噪声注入到输入，对于某些模型而言，向输入添加方差极小的噪声等价于对权重施加范数惩罚

向隐藏单元施加噪声

施加于权重的噪声，可以被解释为与更传统的正则化形式等同，鼓励要学习的函数保持稳定

向输出目标注入噪声，大多数数据集的y标签都有一定错误。错误的y不利于最大化log p(y | x).避免这种情况的一种方法是显式对标签上的噪声进行建模。标签平滑的优势是能够防止模型追求确切概率而不影响模型学习正确分类

半监督学习

生成模型P(x) 未标记样本 或P(x, y) 标记样本

判别模型p(y | x)

多任务学习

通过合并几个任务中的样例（可以视为对参数施加的软约束 ）来提高泛化的一种方式。当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值（如果共享合理），通常会带来更好的泛化能力

提前终止

可以认为提前终止是非常高效的超参数选择算法，训练频数仅是另一个超参数

提前终止可以将优化过程的参数空间限制在初始参数值的小领域内

将训练集分为训练集和验证集，使用提前终止确定训练步数，然后在所有数据上训练

参数绑定和参数共享

表达我们对模型参数适当值 的先验知识，基于相关领域和模型结构方面的知识，得知模型参数之间应该存在一些相关性

若一些任务足够相似（或许具有相似的输入和输出分布），那么这些模型参数应彼此靠近

自然图像有许多统计属性是对转换不变的，CNN通过在图像多个位置共享参数来考虑这个特性。相同的特征（具有相同权重的隐藏单元）在输入的不同位置上计算获得

参数共享显著降低了CNN模型的参数数量，并显著提高了网络的大小而不需要相应地增加训练数据

Bagging和Dropout

Bagging是通过结合几个模型降低泛化误差的技术，主要想法是分别训练几个不同的模型，然后让所有模型表决测试样例的输出。只有当随机抽样的集成成员相互独立地训练好后，才能达到Bagging集成的正则化效果

神经网络中随机初始化的差异、小批量的随机选择、超参数的差异或不同输出的非确定性实现往往足以使得集成中的不同成员具有部分独立的误差

在作为科学论文算法的基准时，它通常是不鼓励使用，因为任何机器学习算法都可以从模型平均中大幅获益（以增加计算和存储为代价）

Dropout可以被认为是集成大量深层神经网络的实用Bagging方法

Dropout的目标是在指数级数量的神经网络上近似这个过程

在Bagging的情况下，所有模型都是独立的，在Dropout的情况下，所有模型共享参数。在Bagging的情况下，每一个模型在其相应训练集上训练到收敛。在Dropout的情况下，通常大部分模型都没有显式地被训练，因为通常父神经网络会很大，以至于到宇宙毁灭都不可能采样完所有子网络。取而代之的是，在单个步骤中我们训练一小部分的子网络，参数共享会使得剩余的子网络也能有好的参数设定

Droput减少了模型的有效容量，为了抵消这种影响，我们必须增大模型规模。不出意外的话，使用Dropout时最佳验证集的误差会低很多，但这是以更大的模型和更多训练算法的迭代次数为代价换来的

只有极少的训练样本可用时（比如低于5000），Dropout不会很有效

通过随机行为训练网络并平均多个随机决定进行预测，实现了一种参数共享的Bagging形式

Dropout不仅仅是训练一个Bagging的集成模型，而且是共享隐藏单元的集成模型，这意味着无论其他隐藏单元是否在模型中，每个隐藏单元必须都能够表现良好。隐藏单元必须准备好进行模型之间的交换和互换。通俗地讲，就是别的隐藏单元随时挂了，我得更努力，才能存活下来

Dropout强大的大部分原因来自施加到隐藏单元的掩码噪声，对输入内容的信息高度智能化、自适应破坏。例如，如果模型学得通过鼻检测脸的隐藏单元hi, 那么丢失hi对应于擦除图像中有鼻子的信息，模型必须学习另一种hi, 如眼睛，嘴巴，耳朵等特征，也能鲁棒鉴别

对抗训练

对抗样本，对抗训练

### 深度模型中的优化

学习和纯优化有什么不同？

机器学习通常是间接作用的，希望通过降低代价函数J(theta)来提高性能度量P.

纯优化最小化目标J本身

理论上，我们希望最小化取自数据生成分布Pdata的期望，然后实际中代价函数写为经验分布的期望（即训练集上的平均）

最小化经验风险：最小化训练集上的期望损失，即用训练集上的经验分布替代真实分布p(x, y)

经验网络最小化：基于最小化平均训练误差的训练过程。经验风险最小化很容易导致过拟合，高容量的模型会简单地记住训练集

小批量算法

机器学习算法和一般优化算法不同，机器学习算法的目标函数通常可以分解为训练样本上的求和。机器学习中的优化算法在计算参数的每一次更新时通常仅使用整个代价函数中一部分项来估计代价函数的期望值。

* N个样本均值的标准差是phi/sqrt(n), 其中phi是样本值真实的标准差，分母sqrt(n)表明使用更多样本来估计梯度的方法的回报是低于线性的。试想，100个样本和10000样本的梯度计算，后者需要100倍计算量，但只降低了10倍的均值标准差。如果能够快速计算梯度估计值，而不是缓慢地计算准确值，那么大多数优化算法会收敛地更快（就总的计算量而言，而不是指更新次数）
* 从小数目样本中获得梯度的统计估计的动机是训练集的冗余，实践中可能会发现大量样本都对梯度做出了非常相似的贡献
* 小批量必须是随机抽取的。因为要求两个连续的梯度估计是互相独立的，因此两个连续的小批量样本也应该是彼此独立的。很多现实的数据集自然排列，从而使得连续的样本之间具有高度相关性。所以在抽取小批量样本前打乱样本顺序。实践中通常将样本顺序打乱一次，然后按照这个顺序存储起来就足够了。
* 第一次遍历时，每个小批量样本都用来计算真实泛化误差的无偏估计； 第二次遍历时，估计将会是有偏的，因为它重新抽取了已经用过的样本
* 除非训练集特别大，通常最是多次遍历训练集。当多次遍历数据集更新时，只有第一遍满足泛化误差梯度的无偏估计。但是，额外的遍历更新当然会由于减小训练误差而得到足够的好处，以抵消其带来的训练误差和测试误差间差距的增加

神经网络优化中的挑战

* Hessian矩阵H的病态，很难合适的学习步长
* 局部极小值

对于实际中感兴趣的网络，是否存在大量代价很高的局部极小值，优化算法是否会碰到这些局部极小值，都是尚未解决的公开问题？

现在学者们都猜想：对于足够大的神经网络而言，大部分局部极小值都具有很小的代价函数，我们能不能找到真正的全局最小点不重要，而是需要在参数空间中找到一个代价很小的点

在神经网络训练中，我们通常不关注某个函数的精确极小点，而只关注将其值下降到足够小以获得一个良好的泛化误差

* 鞍点

鞍点视为代价函数某个横截面上的局部极小点，同时也可以视为代价函数某个横截面上的局部极大点

多类随机函数表现出以下性质：低维空间中，局部极小值很普遍。在更高维空间中，局部极小值很罕见，而鞍点很常见。直觉上可以这么理解：Hessian矩阵在局部极小值得只有正特征值。而在鞍点处，Hessian矩阵则同时具有正负特征值。

牛顿法的目标是寻求梯度为0的点，牛顿法很容易跳进鞍点，梯度下降旨在朝“下坡”移动，而非明确寻求临界点。所以高维空间中鞍点的激增或许解释了在神经网络训练中为什么二阶方法无法成功取代梯度下降

* 梯度消失梯度爆炸

由于变深的结构使模型丧失了学习到先前信息的能力，让优化变得极其困难

假设计算图含一条反复与矩阵W相乘的路径。那么t步后，相当于乘以Wt,

Wt = (V diag(rambda)V-1)t = V diag(rambda)t V-1

若特征值 rambda在量级上大于1，则会爆炸；若小于1则会消失

梯度消失与爆炸问题是指该计算图上的梯度也会因为diag(rambda)t大幅度变化。梯度消失使得我们难以知道参数朝着哪个方向移动能够改进代价函数，而梯度爆炸会使得学习不稳定

几何解释：悬崖

多层神经网络通常存在像悬崖一样的斜率较大区域，这是由于几个较大的权重相乘导致。当参数接近这样的悬崖区域时，梯度下降更新可以使参数弹射得非常远，可能会使大量已完成的优化工作成为无用功。

解决方法：启发式梯度截断，其基本想法源自梯度并没有指明显最佳步长，只说明了在无限小区域内的最佳方向。当传统的梯度下降算法提议更新很大一步时，启发式梯度截断会干涉来减小步长

* 近似梯度

大多数优化算法的先决条件：计算梯度或Hessian矩阵。在实践中，通常这些量会有噪声，甚至是有偏的估计。使用训练样本的小批量来计算梯度

参数初始化策略

* 初始点能够决定算法是否收敛

有些初始点十分不稳定，使得该算法会遭遇数值困难，并完全失败。此外，差不多代价的点可以具有区别极大的泛化误差，初始点也可以影响泛化。

* 初始参数需要在不同单元间“破坏对称性”。

如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须具有不同的初始参数，如果它们具有相同的初始参数，然后应用到确定性损失和模型的确定性学习算法将一直以相同的方式更新这两个单元。最好还是初始化每个单元使其和其他单元计算不同的函数。这或许有助于确保没有输入模式丢失在前向传播的零空间中，没有梯度模式丢失在反向传播的空间中。

* 在高维空间上使用高熵分布来随机初始化

初始化模型的权重为高斯或均匀分布中随机抽取的值。高斯或均匀分布的选择似乎不会有很大的差别

* 关于如何初始化网络，正则化和优化有着非常不同的观点？

优化观点建议权重应该足够大以成功传播信息，矩阵中更大的值在矩阵乘法中有更大的输出。

正则化希望其小一点，如果初始权重太大，那么会在前向传播或反向传播中产生爆炸的值。较大的权重也会产生使得激活函数饱和的值，导致饱和单元的梯度完全丢失。

* 标准初始化normalized initialization

Wi,j ~ U(-sqrt(6/(m+n)), sqrt(6/(m+n)))

* 选择偏置以避免初始化引起太大饱和

例如，我们可能会将ReLU的隐藏单元设为0.1而非0, 以避免ReLU在初始化时饱和。

* 即使在一个不相关的任务上运行监督训练，有时也能得到一个比随机初始化具有更快收敛率的初始值

基本算法

选择哪一个算法，似乎主要 取决于使用者对算法的熟悉程度（以便调节超参数）

随机梯度下降SGD

按照数据生成分布抽取m个小批量（独立同分布的）样本，通过计算它们梯度的均值，可以得到梯度的无偏估计

SGD及相关的小批量亦或更广义的基于梯度优化的在线学习算法，一个重要的性持是每一步更新的计算时间不依赖训练样本数目的多少。即使训练样本数目非常大，它们也能收敛。对于足够大的数据集，SGD可能会在处理整个训练集之前就收敛到最终测试集误差的某个固定容差范围内。

梯度估计：g <- sum(delta L) / m

参数 更新：theta <- theta – lr \* g

动量

旨在加速学习，特别是处理高曲率、小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且继续沿该访问移动

解决了两个问题：Hessian矩阵的病态条件和随机梯度的方差

梯度估计：g <- sum(delta L) / m

速度更新：v <- alpha\*v - lr \* g alpha对应着最大速度1/(1-alpha)倍于梯度下降

参数更新：theta <- theta + v

AdaGrad

具有损失大偏导的参数相应地有一个快速下降的学习率，而具有小偏导的参数在学习率上有相对较小的下降

凸优化快速收敛

梯度估计：g <- sum(delta L) / m

累积平方梯度：r <- r + g.g （点乘）

学习量更新：delta <- -g . lr / (sigma + sqrt(r) ) 点乘

参数更新：theta <- theta + delta

RMSProp (Hinton, 2012)

AdaGrad根据平方梯度的整个历史收缩学习率

修改AdaGrad以在非凸设定下效果更好，

RMSProp使用指数衰减平均以丢弃遥远过去的历史，使其能够在找到凸碗状结构后快速收敛，它就 像一个初始化于该碗状结构的AdaGrad算法实例

梯度估计：g <- sum(delta L) / m

累积平方梯度：r <- p\*r + (1-p)\*g.g p用来控制移动平均的长度范围

学习量更新：delta <- -g . lr / (sigma + sqrt(r) )

参数更新：theta <- theta + delta

Adam (Kingma and Ba, 2014)

= adaptive moments = 结合动量的RMSProp， 通常被认为对超参数的选择相当鲁棒

梯度估计：g <- sum(delta L) / m

更新有偏一阶矩估计：s <- p1\*s + (1-p1) g

更新有偏二阶矩估计：r <- p2\*r + (1-p2)g.g

修正一阶矩的偏差：s’ <- s / (1-p1t)

修正二阶矩的偏差：r’ <- r / (1-p2t)

学习量更新：delta = - lr \* s’/ (sigma + sqrt(r’) )

参数更新：theta <- theta + delta

Batch Normalization



1. Normalizing input (LeCun et al 1998 “Efficient Backprop”)
2. BN: normalizing each layer, for each mini-batch

Pro:

* Greatly accelerate training (set high learning rate)
* Less sensitive to initialization
* Improve regularization (Dropout is not required)

梯度不会再简单地增加隐单元的标准差或均值； 标准化操作会除掉这一操作的影响，归零其在梯度中的元素

优点：批标准化显著地使得模型更易学习，

代价：标准化一个单元的均值和标准差会降低包含该单元的神经网络的表达能力

在线性模型，容易学习的代价是使得底层网络没有用（因为标准化均值 和方差即一阶和二阶统计量，是线性网络可以影响的所有因素）

在具有非线性激活函数的深度网络中，较低层可以进行数据的非线性变换，所以它们仍然是有用的。批标准化仅标准化每个单元的均值和方差，以稳定化学习，但允许单元和单个单元的非线性统计量之间的关系发生变化

推荐做法：批标准化应用于变换后的值XW+b, 而不是输入X，因为一层的输入通常是前一层的非线性激活函数的输出，困此输入的统计量更符合非高斯，而更不服从线性操作的标准化）

在测试阶段，均值和标准差可以被替换为训练阶段收集的运行均值

坐标下降法

当优化问题中的不同变量能够清楚地分成相对独立的组，或是当优化一组变量明显比优化所有变量效率更 高时，坐标下降最有意义

如J(H, W) = …

函数J不是凸的，然而，我们可以将训练算法的输入分成两个集合：字典参数W和编码表示H. 最小化关于这两者之一的任意一组变量的函数都是凸问题。坐标下降法允许我们使用高效的凸优化算法，交替固定H优化W和固定W优化H.

当一个变量的值很大程度地影响另一个变量的最优值时，坐标下降不是一个很好的方法

监督预训练

1. 训练一个浅网络
2. 只保留浅网络的输入到隐藏层，丢弃隐藏层到输出层
3. 将浅网络的输出作为输入训练另一个浅网络

重复以上过程

1. 联合精调所有层

为什么贪心监督预训练会有帮助呢？有助于更好地指导深层结构 的中间层的学习

拓展：迁移学习 （底层特征变化不大，可以用别的任务训练的权重）

FitNets

思路：容易训练的网络 作为很深很窄难训练网络的老师

1. 先训练深度足够低和宽度足够大，容易训练的网络（称为老师网络）
2. 训练学生网络不仅需要预测原任务的输出，还需要预测教师网络中间层的值

尽管一个窄而深的网络似乎比宽而浅的网络更难训练，但窄而深网络的泛化能力可能更好，并且如果其足够窄，参数足够少，那么其计算代价更小。

没有隐藏层的提示，学生网络在训练集和测试集上的实验表现都很差。因而中 间层的提示是有助于训练很难训练的网络方法之一

设计有助于优化的模型

在实践中，选择一族容易优化的模型比使用一个强大的优化算法更重要。神经网络学习在过去30年的大多数进步主要来自改变模型族，而非改变优化过程

现代神经网络的设计选择体现在层之间的线性变换，几乎处处可导的激活函数和大部分定义域都有明显的梯度

层之间的线性路径或是跳跃连接减少了从较低层参数到输出最短路径的长度，因而缓解了梯度消失的问题

延拓法

平滑目标函数，构建更容易的代价函数

逐步求解平滑更少的目标函数来求解全局极小值

延拓法引入的简化目标函数能够消除平坦区域，减少梯度估计的方差，提高Hessian矩阵的条件数，使局部更新更容易计算

课程学习

基于规划学习过程的想法，首先学习简单的概念，然后逐步学习依赖于这些简化概念的复杂概念。

课程学习被证实为与人类教学方式一致，教师刚开始会展示更容易、更典型的示例，然后帮助学习者在不太显然的情况下提炼决策面。

随机课程 可以获得更好的结果，其中容易和困难的示例混合在一起，随机提供给学习者，更难示例（这些具有长期依赖）的平均比例在逐渐上升

### 卷积网络

许多机器学习的库实现的是互相关函数，但是称之为卷积

一个基于核翻转的卷积运算的学习算法所学得的核，是对未进行翻转的算法学得的核的翻转

卷积运算通过三个重要的思想来帮助改进机器学习系统：

稀疏交互sparse interactions (稀疏连接，稀疏权重)

核大小远小于输入的大小

在深度卷积网络中，处在网络深层的单元可能与绝对部分输入是间接交互的

通过卷积核来检测一些小的有意义的特征

参数共享parameter sharing

对于卷积，参数共享的特殊形式使得神经网络层具有对平移等变的性质。

何为等变？如果一个函数满足输入改变，输出也以同样的方式改变这一性质。先变换后卷积 =先卷积后变换

在卷积网络的第一层进行图像的边缘检测是有很用的。相同的边缘或多或少地散落在图像的各处，所以应当对整个图像进行参数共享

卷积对缩放和旋转不等变

等变表示equivariant representation

池化

使用某一位置的相邻输出 的总体统计特征来代替网络在该位置的输出

常用的统计量：最大值，均值，范数，基于距中心像素距离的加权平均函数

池化确保局部平移不变性，对空间区域进行池化产生了平移不变性

使用池化可以看作增加了一个无限强的先验：这一层学得的函数必须具有对少量平移的不变性。

因为池化综合了全部邻居的反馈，这使得池化单元少于探测单元成为可能

处理大小可变的图像的卷积网络，使用具有可变大小但是数量固定的池的池化操作，以便向网络的全连接

池化应对不同大小的输入？对任意大小的输入，池化为m\*m网络的输出

我们相对不同大小的图像进行分类时，分类层的输入必须是固定的大小，而这通常通过调整池化区域的偏置大小来实现，这样分类层总是能接收到相同数量的统计特征而不管最初的输入大小了。

# 模型

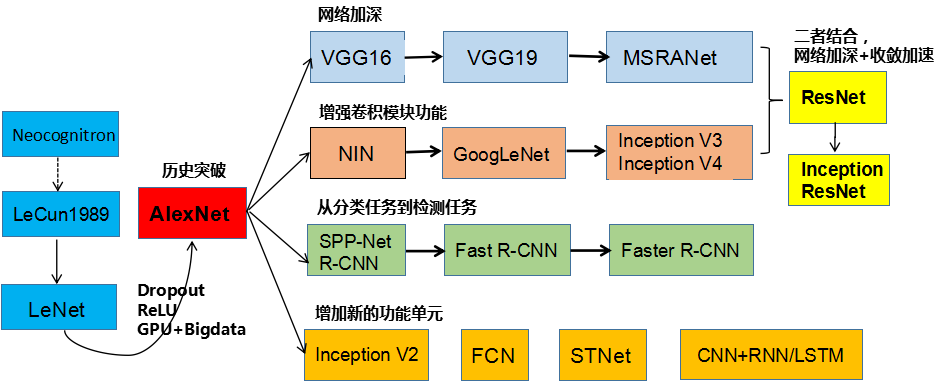
全连接网络：下层的节点与上层的每个节点相连，且每个节点各自使用一套参数。在全连接的网络中，假如k层有n个节点，k+1层有m个节点，则一共有n\*m个连接；每个连接都有一个参数，外加每个k+1层节点有一个bias，则共有n\*m + m个训练参数

卷积神经网络

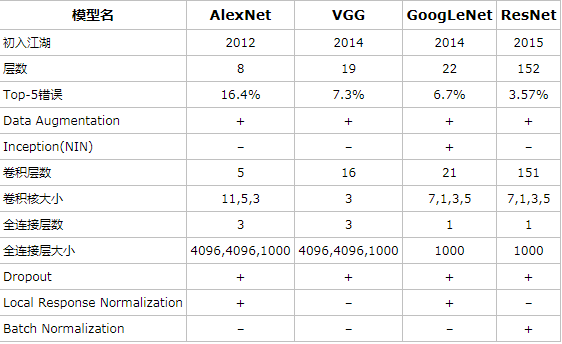
采用局部连接和参数共享的方式连接网络

局部连接是相对于普通神经网络的全连接而言的，是指这一层的某个节点只与上一层的部分节点相连。参数共享是指一层中多个节点的连接共享相同的一组参数。

对于一个卷积神经网络，假如该网络的第k层有n个节点，k+1层为卷积层且有m个节点，则k+1层的每个节点只与k层的部分节点相连，此处假设只与k层的i个节点相连（局部连接）；另外k+1层的每个节点的连接共享相同的参数、相同的bias（参数共享）。这样该卷积神经网络的第k、k+1层间共有m\*i个连接、i+1个参数

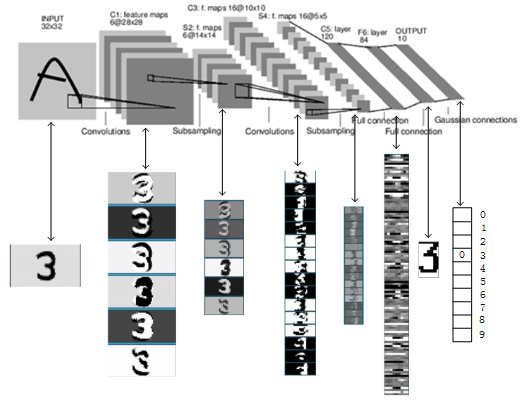


ILSVRC竞赛（ImageNet数据集）



## LeNet

LeNet-5是Yann LeCun在1998年设计的用于手写数字识别的卷积神经网络，是早期卷积神经网络中最有代表性的实验系统之一



LeNet-5共有5层（不包括输入层），输入图像为32\*32大小。这要比Mnist数据库（一个公认的手写数据库）中最大的字母还大。这样做的原因是希望潜在的明显特征如笔画断电或角点能够出现在最高层特征监测子感受野的中心。

每个层有多个Feature Map，**每个Feature Map通过一种卷积滤波器提取输入的一种特征**，然而每个Feature Map有多个神经元。Feature map内的神经元共享权重

C1层：卷积层，由6个特征图Feature Map构成。特征图中每个神经元与输入中5\*5的邻域相连。特征图的大小为28\*28。C1有156个可训练参数（每个滤波器5\*5=25个unit参数和一个bias参数，一共6个滤波器，共(5\*5+1)\*6=156个参数），共156\*(28\*28)=122,304个连接。

S2层：下采样层（为什么是**下采样？利用图像局部相关性的原理，对图像进行子抽样，可以减少数据处理量同时保留有用信息，最终是为了降低网络训练参数及模型的过拟合程度**），有6个14\*14的特征图。特征图中的每个单元与C1中相对应特征图的2\*2邻域相连接。S2层每个单元的4个输入相加，乘以一个可训练参数，再加上一个可训练偏置。结果通过sigmoid函数计算。可训练系数和偏置控制着sigmoid函数的非线性程度。如果系数比较小，那么运算近似于线性运算，亚采样相当于模糊图像。如果系数比较大，根据偏置的大小亚采样可以被看成是有噪声的“或”运算或者有噪声的“与”运算。每个单元的2\*2感受野并不重叠，因此S2中每个特征图的大小是C1中特征图大小的1/4（行和列各1/2）。S2层有12个可训练参数和5880个连接。

S-层可看作是模糊滤波器，起到二次特征提取的作用。隐层与隐层之间空间分辨率递减，而每层所含的平面数递增，这样可用于检测更多的特征信息

C3层：卷积层，它同样通过5x5的卷积核去卷积层S2，然后得到的特征map就只有10x10个神经元，但是它有16种不同的卷积核，所以就存在16个特征map了。这里需要注意的一点是：**C3中的每个特征map是连接到S2中的所有6个特征map的，表示本层的特征map是上一层提取到的特征map的不同组合**。（看到没有，这里是组合，就像之前聊到的人的视觉系统一样，底层的结构构成上层更抽象的结构，例如边缘构成形状或者目标的部分）。

S4层：下采样层，由16个5\*5大小的特征图构成。特征图中的每个单元与C3中相应特征图的2\*2邻域相连接，跟C1和S2之间的连接一样。S4层有32个可训练参数（每个特征图1个因子和一个偏置）和2000个连接。

C5层：卷积层，有120个特征图。每个单元与S4层的全部16个单元的5\*5邻域相连。由于S4层特征图的大小也为5\*5（同滤波器一样），故C5特征图的大小为1\*1：这构成了S4和C5之间的全连接。

F6层： 有84个单元，与C5层全相连。有10164个可训练参数

输出层：由欧式径向基函数（Euclidean Radial Basis Function）单元组成，每类一个单元，每个有84个输入。换句话说，每个输出RBF单元计算输入向量和参数向量之间的欧式距离。输入离参数向量越远，RBF输出的越大。一个RBF输出可以被理解为衡量输入模式和与RBF相关联类的一个模型的匹配程度的惩罚项。用概率术语来说，RBF输出可以被理解为F6层配置空间的高斯分布的负log-likelihood。

## AlexNet

AlexNet 之所以能够成功，深度学习之所以能够重回历史舞台，原因在于：

* 非线性激活函数：**ReLU**

Sigmoids saturate and kill gradients. sigmoid 有一个非常致命的缺点，当输入非常大或者非常小的时候，会有饱和现象，这些神经元的梯度是接近于0的。如果你的初始值很大的话，梯度在反向传播的时候因为需要乘上一个sigmoid 的导数，所以会使得梯度越来越小，这会导致网络变的很难学习。

主要是因为它是linear，而且 non-saturating（因为ReLU的导数始终是1），相比于 sigmoid/tanh，ReLU 只需要一个阈值就可以得到激活值，而不用去算一大堆复杂的运算。

* **防止过拟合的方法**：Dropout，Data augmentation, LRN归一化层的使用

结合预先训练好的许多不同模型，来进行预测是一种非常成功的减少测试误差的方式（Ensemble）。但因为每个模型的训练都需要花了好几天时间，因此这种做法对于大型神经网络来说太过昂贵。

然而，AlexNet 提出了一个非常有效的模型组合版本，它在训练中只需要花费两倍于单模型的时间。这种技术叫做Dropout，它做的就是以0.5的概率，将每个隐层神经元的输出设置为零。以这种方式“dropped out”的神经元既不参与前向传播，也不参与反向传播。

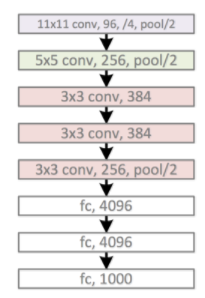
所以每次输入一个样本，就相当于该神经网络就尝试了一个新的结构，但是所有这些结构之间共享权重。因为神经元不能依赖于其他特定神经元而存在，所以这种技术降低了神经元复杂的互适应关系。

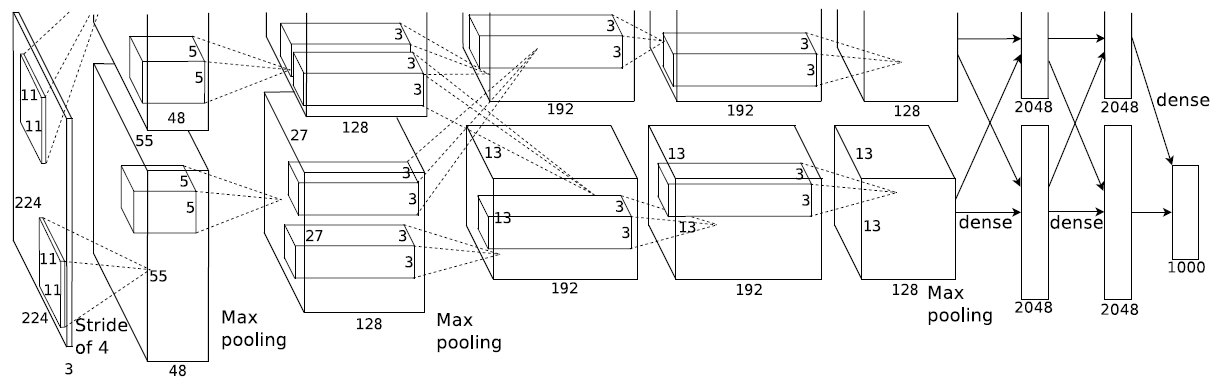
正因如此，网络需要被迫学习更为鲁棒的特征，这些特征在结合其他神经元的一些不同随机子集时有用。在测试时，我们将所有神经元的输出都仅仅只乘以0.5，对于获取指数级dropout网络产生的预测分布的几何平均值，这是一个合理的近似方法。

* **大数据训练**：百万级ImageNet图像数据
* 其他：**GPU**实现

AlexNet 优势在于：网络增大（5个卷积层+3个全连接层+1个softmax层），同时解决过拟合（dropout，data augmentation，LRN），并且利用多GPU加速计算

架构：





## VggNet-19

3\*3 conv 64 + 3\*3 conv 64 + pool/2

3\*3 conv 128 + 3\*3 conv 128 + pool/2

3\*3 conv 256 + 3\*3 conv 256 +3\*3 conv 256 + 3\*3 conv 256, pool/2

3\*3 conv 512 + 3\*3 conv 512 +3\*3 conv 512 + 3\*3 conv 512, pool/2

3\*3 conv 512 + 3\*3 conv 512 +3\*3 conv 512 + 3\*3 conv 512, pool/2

fc 4096

fc 4096

fc 1000

VGGNet-19探索了卷积网络的**深度与其性能**之间的关系，证明使用很小的卷积（3\*3），增加网络深度可以有效提升模型的效果，而且VGGNet对其他数据集具有很好的泛化能力。

VGGNet-19的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（3\*3）和最大池化尺寸（2\*2）

* 卷积网络的参数量不多，因为参数量主要都消耗在全连接层。不过训练比较耗时的部分依然是卷积，因其计算量比较大。
* 两个连续的3\*3的卷积相当于5\*5的感受野，三个相当于7\*7。

使用三个3\*3卷积而不是一个7\*7的卷积的优势有两点：

一、**3个3\*3的卷积层拥有比1个7\*7的卷积层更多的非线性变换**（前者可以使用三次ReLU激活函数，而后者只有一次），使得CNN对特征的学习能力更强。

二、**3个3\*3卷积比1个7\*7卷积层更少的参数量**，只有后者的(3\*3\*3)/(7\*7)=55%。

* **1\*1卷积层主要是为了增加决策函数的非线性**，而不影响卷积层的感受野。虽然1\*1的卷积操作是线性的，但是ReLu增加了非线性。

## GoogLeNet

文章提出获得高质量模型最保险的做法就是**增加模型的深度（层数）或者是其宽度（层核或者神经元数）**，但是这里一般设计思路的情况下会出现两个缺陷

1.参 数太多，容易过拟合，若训练数据集有限；

2.网络越大计算复杂度越大，难以应用；

3.网络越深，梯度越往后穿越容易消失，难以优化模型）。

GoogLeNet的主要思想就是围绕这两个思路去做的：

1.深度，层数更深，文章采用了**22层**，为了避免上述提到的梯度消失问题，googlenet巧妙的在不同深度处增加了两个loss来保证梯度回传消失的现象。

2.宽度，增加了多种核 1x1，3x3，5x5，还有直接max pooling的，为了避免计算量大，在3x3和5x5前，max pooling后分别加上了1x1的卷积核起到了降低feature map厚度的作用。

架构图：Inception Architecture

Input

Conv (7\*7+2s) + MaxPool (3\*3+2s) + LocalRespNorm

Conv (1\*1+1v) + Conv (3\*3+1s) + LocalRespNorm + MaxPool(3\*3+2s)

Inception Module

Inception Module

MaxPool(3\*3+2s)

Inception Module

Inception Module

AveragePool (5\*5 + 3v)

Conv (1\*1 + 1s) + FC + FC

SoftmaxActivation

Inception Module

Inception Module

Inception Module

AveragePool (5\*5 + 3v)

Conv (1\*1 + 1s) + FC + FC

SoftmaxActivation

Inception Module

Inception Module

AveragePool (7\*7 + 1v)

FC

SoftmaxActivation

改进一：**拥有多个不同尺度的kernels，每一个尺度的kernel会学习不同的特征，融合不同尺度的特征给下一层**

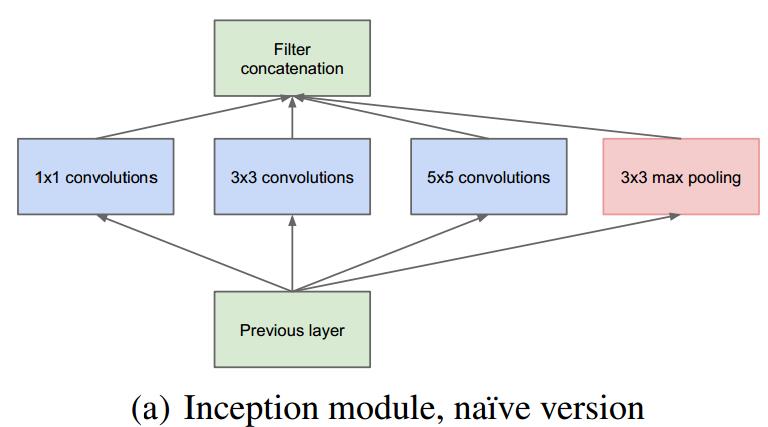
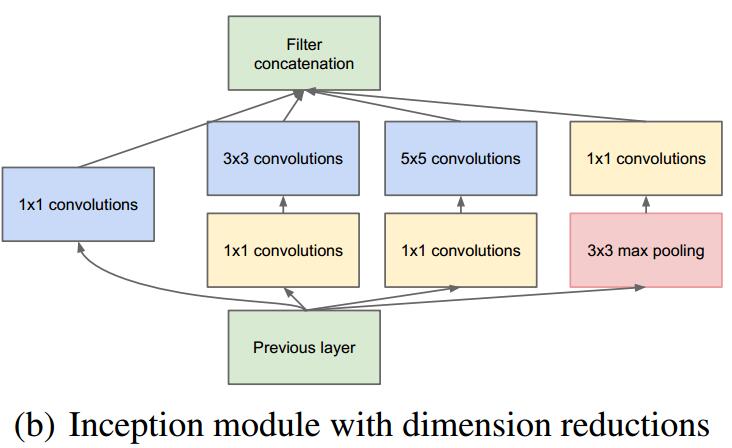
改进二：**3x3, 5x5的卷积核计算量大，巨量参数容易产生过拟合，采用1x1卷积核来进行降维**。

例如：上一层的输出为100x100x128，经过具有256个输出的5x5卷积层之后(stride=1，pad=2)，输出数据为100x100x256。其中，卷积层的参数为128x5x5x256。假如上一层输出先经过具有32个输出的1x1卷积层，再经过具有256个输出的5x5卷积层，那么最终的输出数据仍为为100x100x256，但卷积参数量已经减少为128x1x1x32 + 32x5x5x256，大约减少了4倍。

改进三：**用average pooling代替全连接层，进一步压缩网络参数**

改进四：**为了避免梯度消失，网络额外增加了2个辅助的softmax用于向前传导梯度**

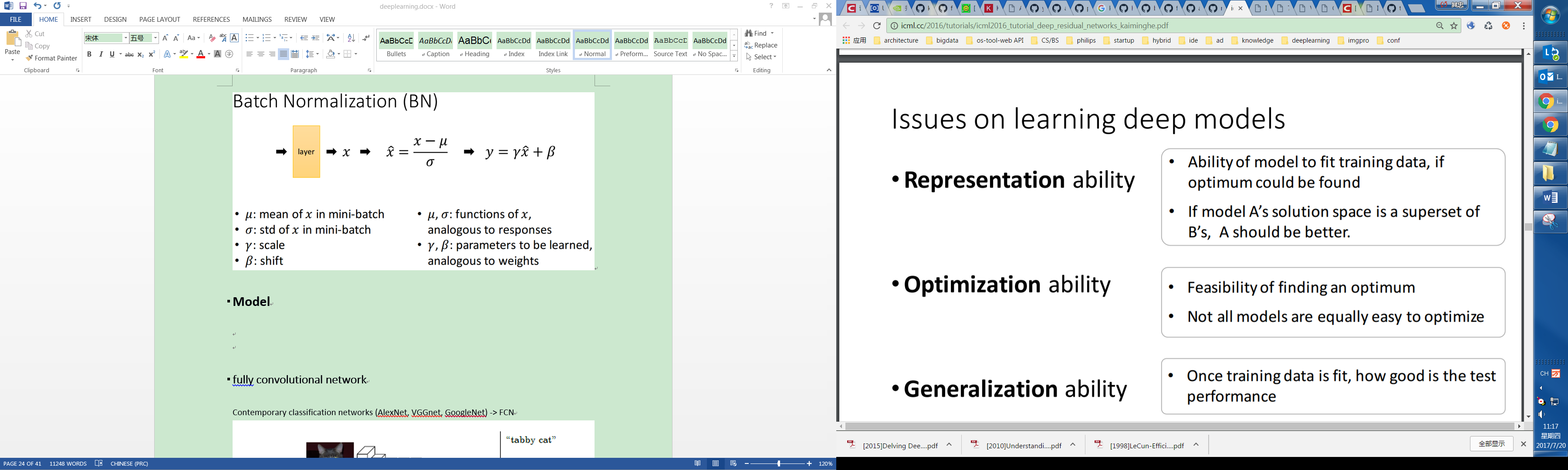
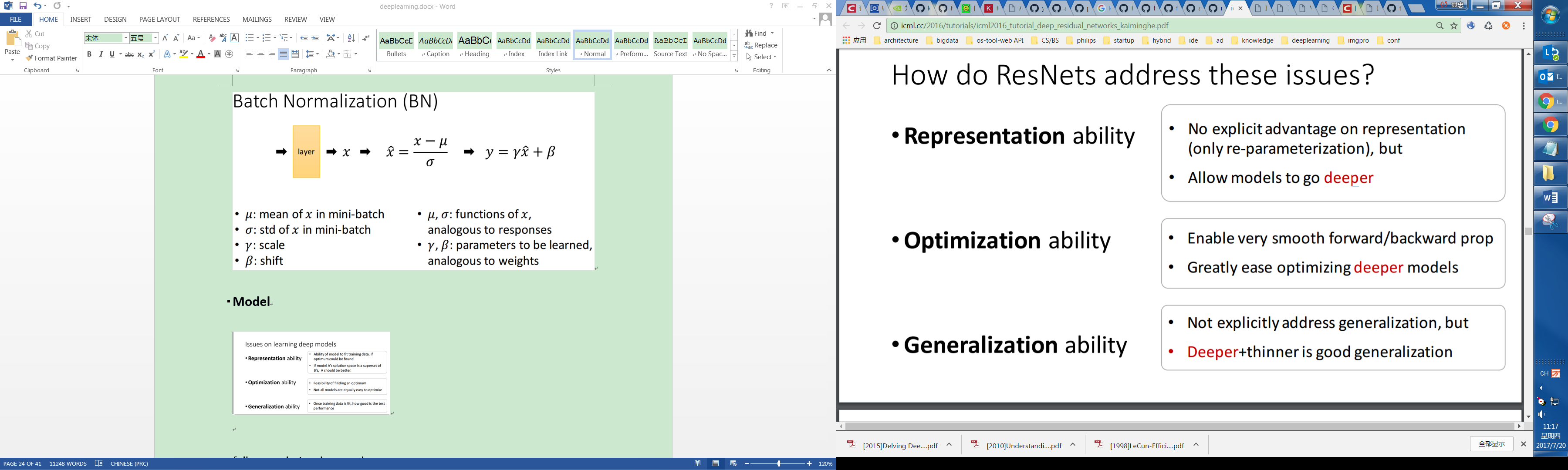
性能与VggNet接近，但计算效率高于VggNet

Inception module特点

* 采用不同大小的卷积核意味着不同大小的感受野，最后拼接意味着不同尺度特征的融合；
* 之所以卷积核大小采用1、3和5，主要是为了方便对齐。设定卷积步长stride=1之后，只要分别设定pad=0、1、2，那么卷积之后便可以得到相同维度的特征，然后这些特征就可以直接拼接在一起了；
* 文章说很多地方都表明pooling挺有效，所以Inception里面也嵌入了。

## ResNET

深度网络自然的整合了低中高不同层次的特征，并且使用端到端的多层次分类，特征的“层次”可以靠加深网络层数来丰富

然而加深网络会带来如下三大类问题：

* 反向传播梯度消失
* 前向传播信息量减少
* 训练时间加长

梯度消失/爆炸的问题，很大程度上可以通过标准的初始化和正则化层来基本解决，确保几十层的网络能够收敛（用SGD+反向传播）

更深层的网络的收敛问题时，退化问题就暴露了：随着神经网络深度的增加，精确度开始饱和（这是不足为奇的），然后会迅速的变差

训练精度的退化表明，不是所有的系统都同样容易优化

例子：

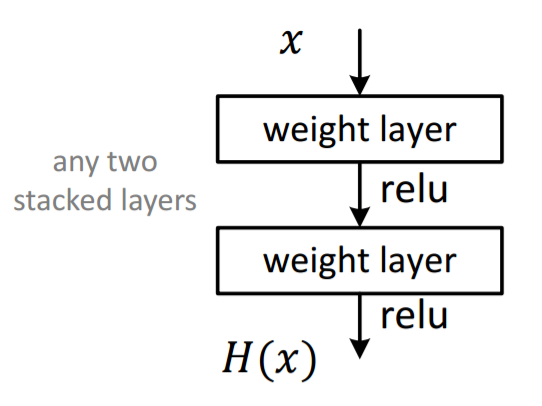
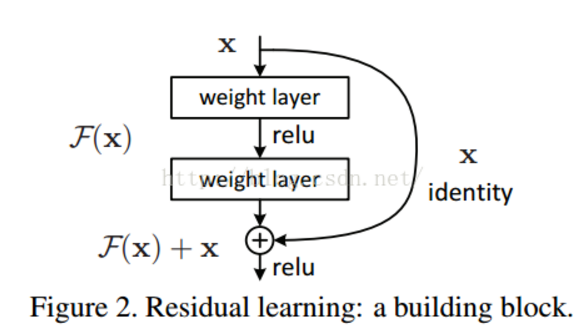
理论上来说，一个层数越多的神经网络，那么它所能拟合的函数就越复杂，得出的错误率就应该越小. 举例：如果在一个浅层网络A上叠加几层layer形成网络B，如果这些新添加的layer是Identity mapping（权值矩阵全是单位矩阵？），那么网络B性能至少不会比A差。但是实际实验结果却显示网络越深，性能越差，这种现象称为**性能退化问题**。性能退化问题暗示多个非线性网络层近似单位映射有困难。

如何解决层次比较深的时候无法训练的问题？

**主要的创新是残差网络。思想借鉴Highway Network的网络相当于旁边专门开个通道使得输入可以直达输出，而优化的目标由原来的拟合输出H(x)变成输出和输入的差H(x)-x**，其中H(X)是某一层原始的的期望映射输出，x是输入

ResNet的原理如下：

假设用几层网络去逼近一个复杂的非线性映射H(x),那么同样可以用这几层网络去逼近它的**residual function（残差映射）**: F(x) = H(x) – x, 而且猜想优化residual mapping要比直接优化H(x)简单

即原最优解映射H(x)=F(x)+x。

如果这个Identity项是最优的，那么旁边的非线性层的参数应该全部为0，然后一层Identity Mapping就代表了最优函数

但是通常情况下，这个x不是最优的，即Identity Mapping接近于最优函数，easier to find small fluctuations。他对于更深层的网络传递到后来的误差就会越小。

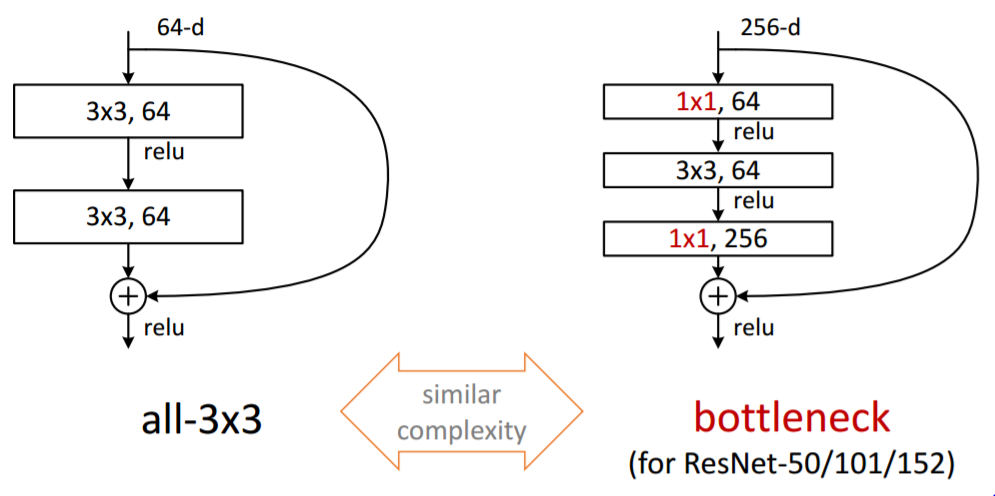
在实际过程中，这H(x)等式右边的这两个分量是可以加上参数的。网络中残差的表达式可以统一写成如下公式：y = F(X, {Wi}) + WsX，其中Ws 只有在feature map维度不同的时候才用到，可以是pad零，也可以是1x1卷积核，即负责将二者的维数变为相同。

网络设计：

**Keep it simple**

VGG-style: all 3\*3 conv, spatial size/2 => filters \* 2, simple design just deep

Other remarks: no max pooling, no hidden fc, no dropout



还记得吗？两个1\*1 conv 等效3\*3 conv， 而且1\*1 conv可以变换维数

**Keep the shortest pass** as smooth as possible

• by making h and f identity

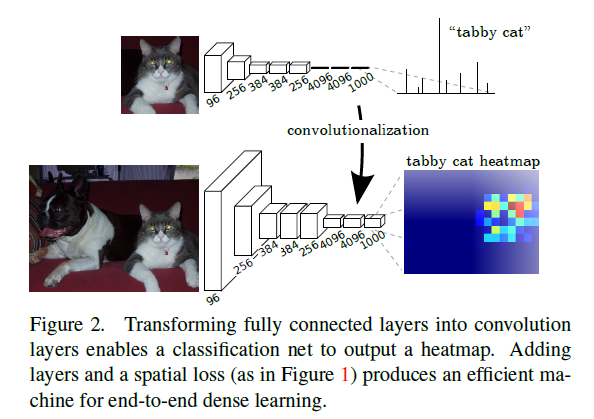
• forward/backward signals directly flow through this path

• Features of any layers are additive outcomes

## fully convolutional network

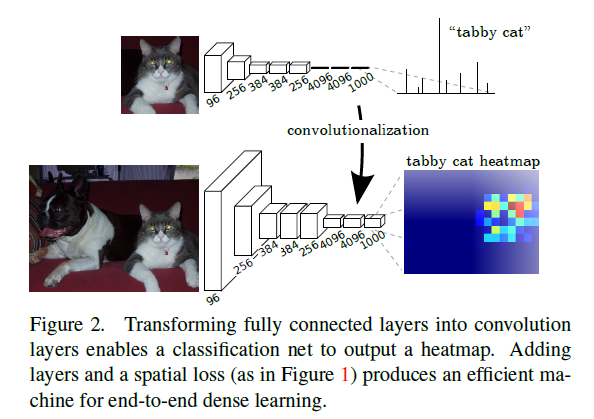
CNN: Contemporary classification networks (AlexNet, VGGnet, GoogleNet)

在一般的卷积神经网络中,一般结构都是前几层是卷积层加池化,最后跟2-3层的全连接层,输出分类结果,如下图所示:



这个结构就是AlexNet的结构,用来进行ImageNet中的图片分类,最后一层是一个输出为1000\*1向量的全连接层,因为一共有1000个类,向量中的每一维都代表了当前类的概率,其中tabby cat的概率是最大的.

而在全卷积神经网络中,没有了全连接层,取而代之的是卷积层,如下图所示:



最后一层输出的是1000个二维数组,其中每一个数组可以可视化成为一张图像,图中的每一个像素点的灰度值都是代表当前像素点属于该类的概率,比如在这1000张图像中,取出其中代表tabby cat的概率图,颜色从蓝到红,代表当前点属于该类的概率就越大.

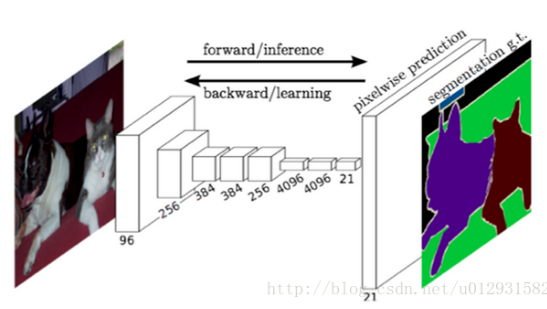
可以看出FCN与CNN之间的区别就是把最后几层的全连接层换成了卷积层,这样做的好处就是能够进行dense prediction.

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：



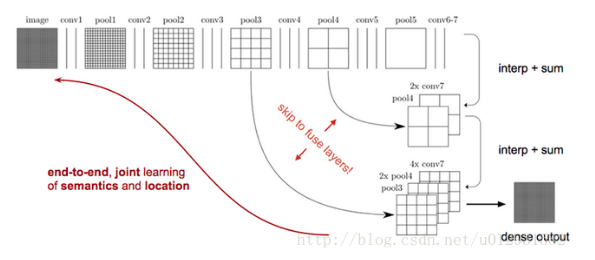
（1）对传统网络如AlexNet，**VGG等的最后全连接层变成卷积层**。

（2）**对中间得到的feature map做bilinear上采样**，就是反卷积层, 使它恢复到输入图像相同的尺寸

2）如何refine，得到更好的结果？

在进行语义分割的时候,需要解决的一个重要问题就是,如何把定位和分类这两个问题结合起来,毕竟语义分割就是进行逐个像素点的分类,就是把where和what两个问题结合在了一起进行解决.

在前面几层卷积层,分辨率比较高,像素点的定位比较准确,后面几层卷积层,分辨率比较低,像素点的分类比较准确,所以为了更加准确的分割,需要把前面高分辨率的特征和后面的低分辨率特征结合起来.



如上图所示，对原图像进行卷积conv1、pool1后原图像缩小为1/2；之后对图像进行第二次conv2、pool2后图像缩小为1/4；接着继续对图像进行第三次卷积操作conv3、pool3缩小为原图像的1/8，此时保留pool3的featureMap；接着继续对图像进行第四次卷积操作conv4、pool4，缩小为原图像的1/16，保留pool4的featureMap；最后对图像进行第五次卷积操作conv5、pool5，缩小为原图像的1/32，然后把原来CNN操作中的全连接变成卷积操作conv6、conv7，图像的featureMap数量改变但是图像大小依然为原图的1/32,此时进行32倍的上采样可以得到原图大小,这个时候得到的结果就是叫做FCN-32s.

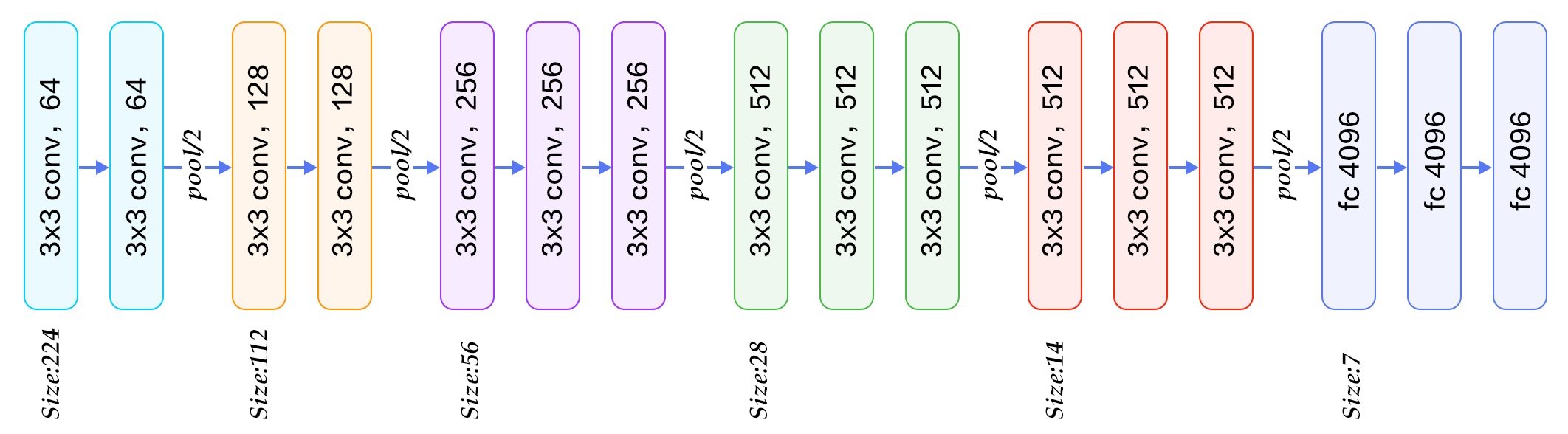
这个时候可以看出,FCN-32s结果明显非常平滑,不精细. 针对这个问题,作者采用了combining what and where的方法,具体来说,就是在FCN-32s的基础上进行fine tuning,把pool4层和conv7的2倍上采样结果相加之后进行一个16倍的上采样,得到的结果是FCN-16s.

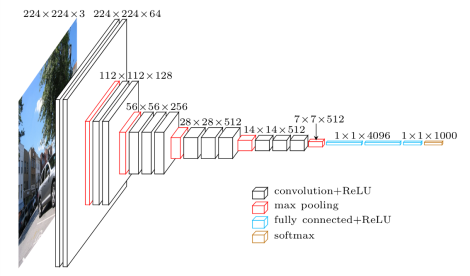
之后在FCN-16s的基础上进行fine tuning,把pool3层和2倍上采样的pool4层和4倍上采样的conv7层加起来,进行一个8倍的上采样,得到的结果就是FCN-8s.

Code:

github.com/aurora95/Keras-FCN

分类的Vgg16如何迁移学习至FCN-Vgg16?





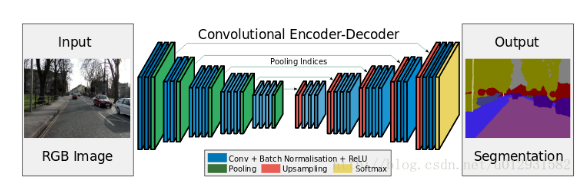
Vgg16

|  |  |  |
| --- | --- | --- |
| Layer | weight | output |
| Conv3\_3\_64 | 3\*3\*3\*64 | 224\*224\*64 |
| Conv3\_3\_64 | 3\*3\*64\*64 | 同上 |
|  | | |
| pooling |  | 112\*112\*64 |
| Conv3\_3\_128 | 3\*3\*64\*128 | 112\*112\*128 |
| Conv3\_3\_128 | 3\*3\*128\*128 | 同上 |
|  | | |
| pooling |  | 56\*56\*128 |
| Conv3\_3\_256 | 3\*3\*128\*256 | 56\*56\*256 |
| Conv3\_3\_256 | 3\*3\*256\*256 | 同上 |
| Conv3\_3\_256 | 同上 | 同上 |
|  | | |
| Pooling |  | 28\*28\*256 |
| Conv3\_3\_512 | 3\*3\*256\*512 | 28\*28\*512 |
| Conv3\_3\_512 | 3\*3\*512\*512 | 同上 |
| Conv3\_3\_512 | 同上 | 同上 |
|  | | |
| Pooling |  | 14\*14\*512 |
| Conv3\_3\_512 | 同上 | 同上 |
| Conv3\_3\_512 | 同上 | 同上 |
| Conv3\_3\_512 | 同上 | 同上 |
|  | | |
| Pooling |  | 7\*7\*512 |
| fc1 | 25088(=7\*7\*512)\*4096 | 4096 |
| fc2 | 4096\*4096 | 4096 |
| Fc3 | 4096\*1000 | 1000 |

FCN\_Vgg16

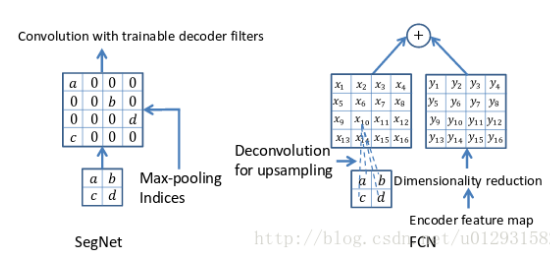
|  |  |  |
| --- | --- | --- |
| Pooling |  | 7\*7\*512 |
| Conv7\_7\_4096 | 7\*7\*512\*4096 | 7\*7\*4096 |
| Conv1\_1\_4096 | 1\*1\*4096\*4096 | 7\*7\*4096 |
| Conv1\_1\_1000 | 1\*1\*4096\*1000 | 7\*7\*1000 |

## SegNet



可以看出,整个结构就是一个encoder和一个decoder.前面的encoder就是采用的vgg-16的网络结构,而decoder和encoder基本上就是对称的结构.

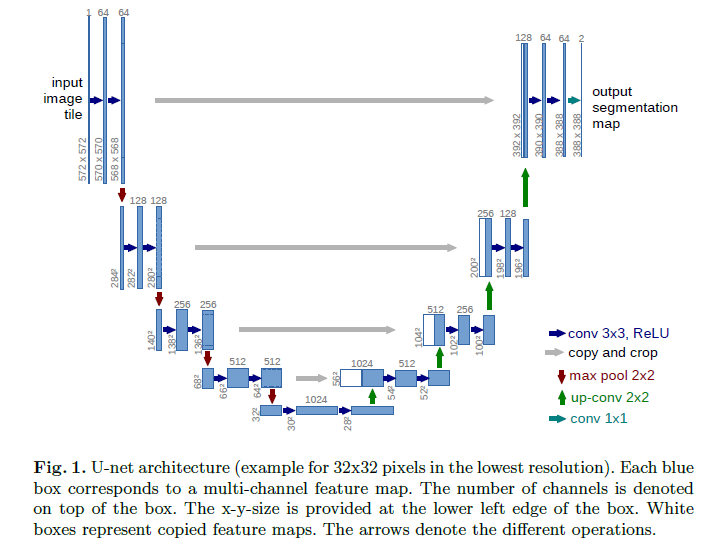
SegNet和FCN最大的不同就在于decoder的upsampling方法,上图结构中,注意,前面encoder每一个pooling层都把pooling indices保存,并且传递到后面对称的upsampling层. 进行upsampling的过程具体如下:



左边是SegNet的upsampling过程,就是把feature map的值 abcd, 通过之前保存的max-pooling的坐标映射到新的feature map中,其他的位置置零.

右边是FCN的upsampling过程,就是把feature map, abcd进行一个反卷积,得到的新的feature map和之前对应的encoder feature map 相加

## 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)

## 对抗样本和对抗网络

lancezhange.com/2015/11/19/adversarial-samples/

m.blog.csdn.net/liuxiao214/article/details/73485333

$ git clone <https://github.com/bstriner/keras-adversarial.git>

github.com/GKalliatakis/Delving-deep-into-GANs 资料大全