



南京邮电大学  
Nanjing University of Posts and Telecommunications

# Python程序设计 (混合式)



for



杨尚东

南京邮电大学计算机学院，数据科学与工程系

[shangdongyang.github.io](https://shangdongyang.github.io)

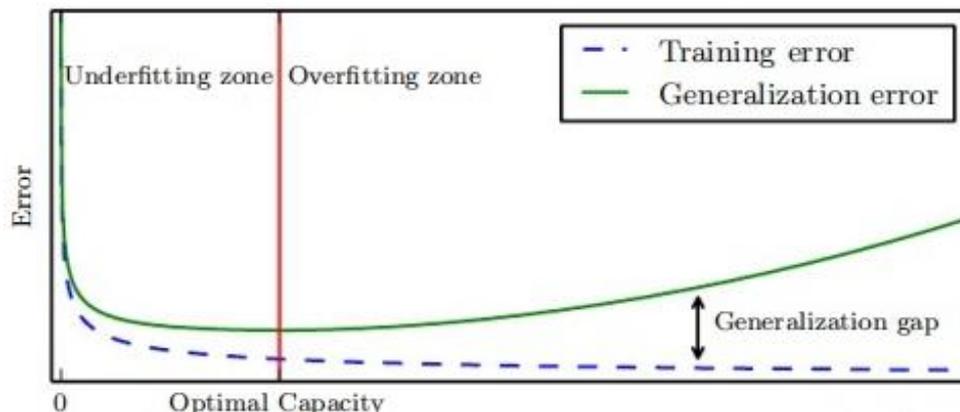
2024/12/13

# 目录

- 神经网络简介
- TensorFlow
- Keras
- 全连接神经网络及Keras实现
- 全连接神经网络的自定义编码实现
- 卷积神经网络及TensorFlow实现
- AlexNet编码实现

# 没有免费的午餐 (No free lunch)

- 在训练模型的过程中，对于有参模型而言，选择模型的容量对应着可以得到的假设空间。假若模型容量不够大，或者没有涵盖到真实的目标函数  $f$ ，就会导致欠拟合
- 反之，若模型的容量过大（自由度过高），就会产生过拟合情况

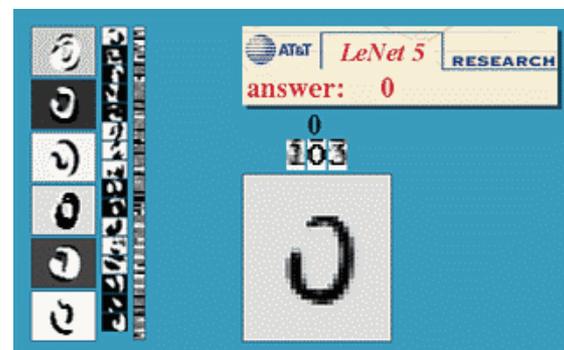
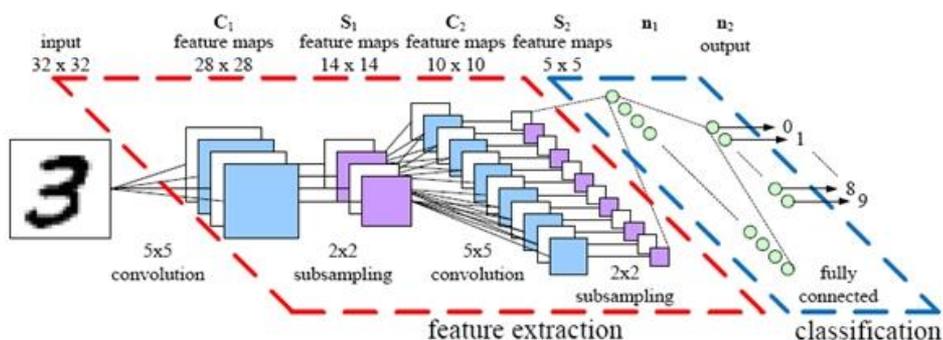


## □ No Free Lunch Theorem

- ✓ 没有一种通用的学习算法可以在各种任务中都有很好的表现
- ✓ 一种算法（算法A）在特定数据集上的表现优于另一种算法（算法B）的同时，一定伴随着算法A在另外某一个特定的数据集上有着不如算法B的表现

# 神经网络简介

- 如何分类图片？
- 神经网络：通过从样本中学习特征和标签间的关系，便可以根据新样本的特征计算其标签



# 原因1：大数据

□ 随着信息化、互联网的高速发展，数据量爆发式地增长，数据维度越来越丰富

□ ImageNet

- ✓ ImageNet项目是一个用于视觉对象识别软件研究的大型可视化数据库。超过1400万的图像URL被ImageNet手动注释，以指示图片中的对象



- 互联网数据
- 用户行为数据
- 结构数据

数据量大

- 用户属性
- 商品维度
- 时间维度
- 多媒体
- ...

维度高

- 图像
- 视频
- 文本
- 语音
- ...

类型多样

# 原因2：运算能力提升

- 摩尔定律：集成电路上可以容纳的晶体管数目在大约每经过18个月便会增加一倍

2013 780Ti

流处理器: 512个
核心频率: 1206MHz
显存频率: 7000MHz
显存规格: 2G GDDR5
显存位宽: 128bit
输出接口: DVI-D/DP/HDMI



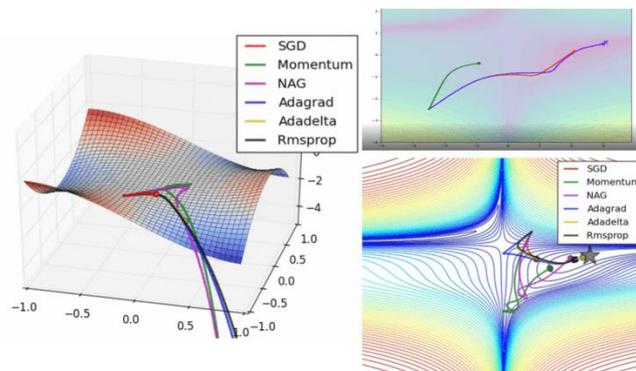
产品参数	
显示核心	GeForce RTX 3090
CUDA数量	10496
核心频率	1695Mhz
一键超频	1785Mhz
显存容量	24GB
显存类型	GDDR6X
显存位宽	384Bit
电源接口	3*8Pin
推荐电源	750W及以上
散热	三风扇散热+6*8mm热管
产品尺寸	323*158*60mm
输出接口	3*DP+HDMI

GeForce RTX 4090	
Architecture	Ada (TSMC N4)
GPU	AD102-300
Board Number	PG139-SKU330
SMs	126 -> <b>128</b>
CUDA Cores	16128 -> <b>16384</b>
CUDA vs Predecessor	<b>+56%</b> (vs 3090)
Memory	24 GB G6X
Memory Bus	384-bit
Memory Speed	21 Gbps
TDP	-450W
Launch Date	September-October 2022

# 原因3：神经网络设计与优化方法

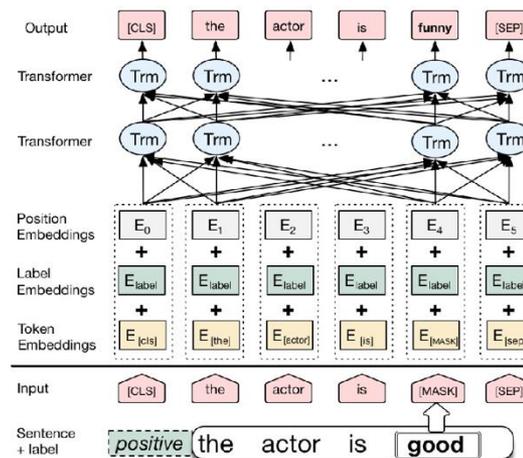
## □ 优化方法

- ✓ 神经网络的目标函数是非凸的



## □ 网络结构

- ✓ 面对大量的数据，需要更复杂的网络结构



# 神经网络简介

- 以图像分类为例，给定若干个 $N \times N$ 维像素矩阵组成的图像样本集，其中 $s$ 为 $S$ 中的一个样本， $x$ 为样本 $s$ 的特征。这里将像素作为图像样本的特征。神经网络根据样本学习得到一个函数 $f()$ ，该函数的值便是当前函数对图片样本的分类结果，公式如下：

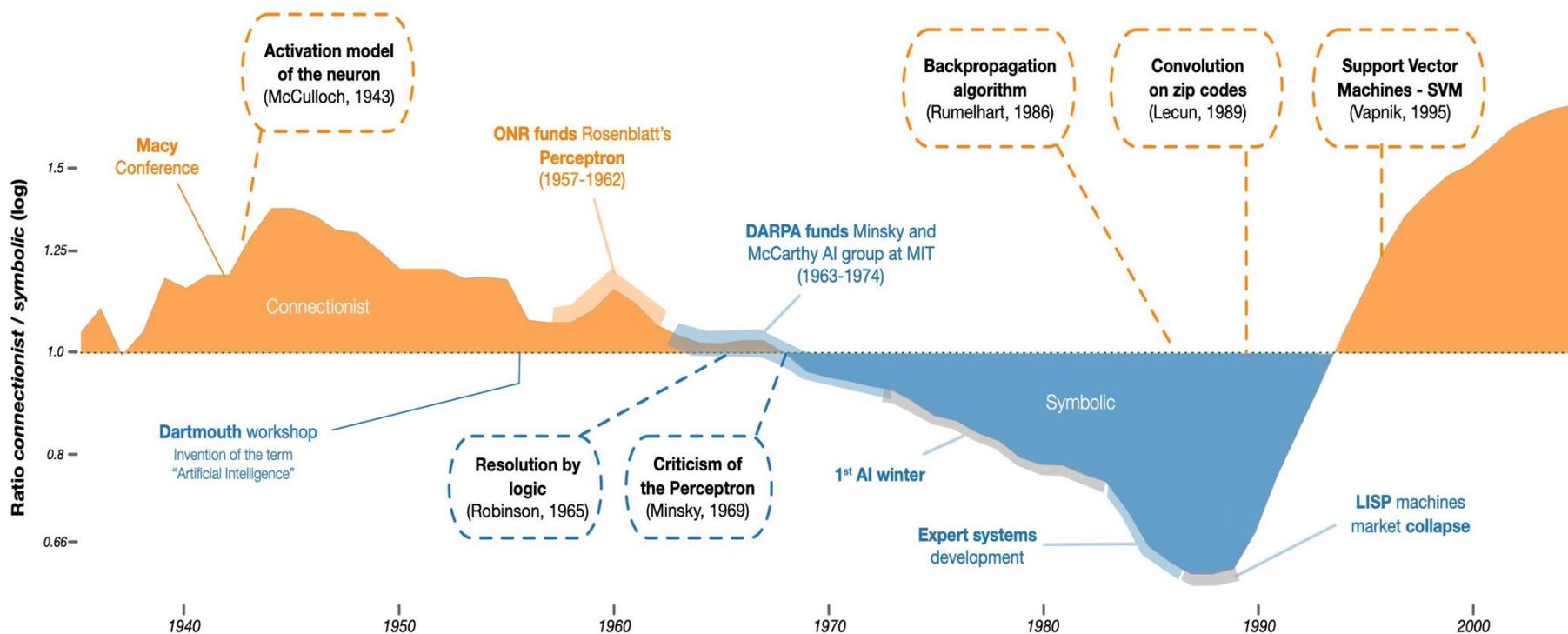
$$\hat{y} = f(x) = f(Wx + \theta)$$

- 该公式就是需要学习的模型。其中， $f()$ 为激活函数， $W$ 为分类函数中的投影矩阵， $\theta$ 为偏置项。神经网络要学习和调整的参数为 $W$ 和 $\theta$ 。训练神经网络的过程就是在寻找一条合适的曲线的过程，使得定义的目标损失函数最小。

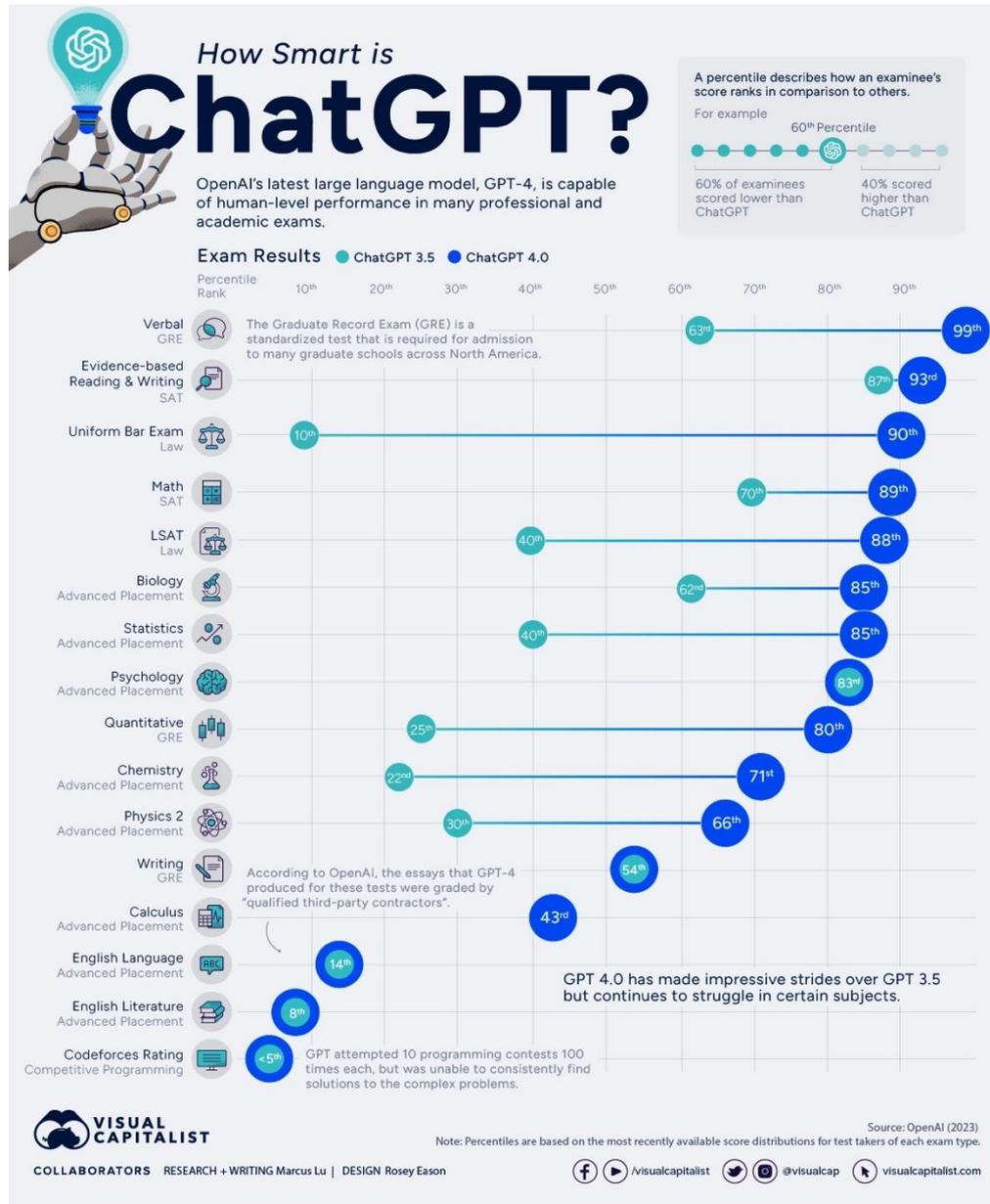
## □ 特点

- ✓ 模型参数多、容量大、非线性
- ✓ 高维特征处理能力
- ✓ 更容易过拟合

# 神经网络发展

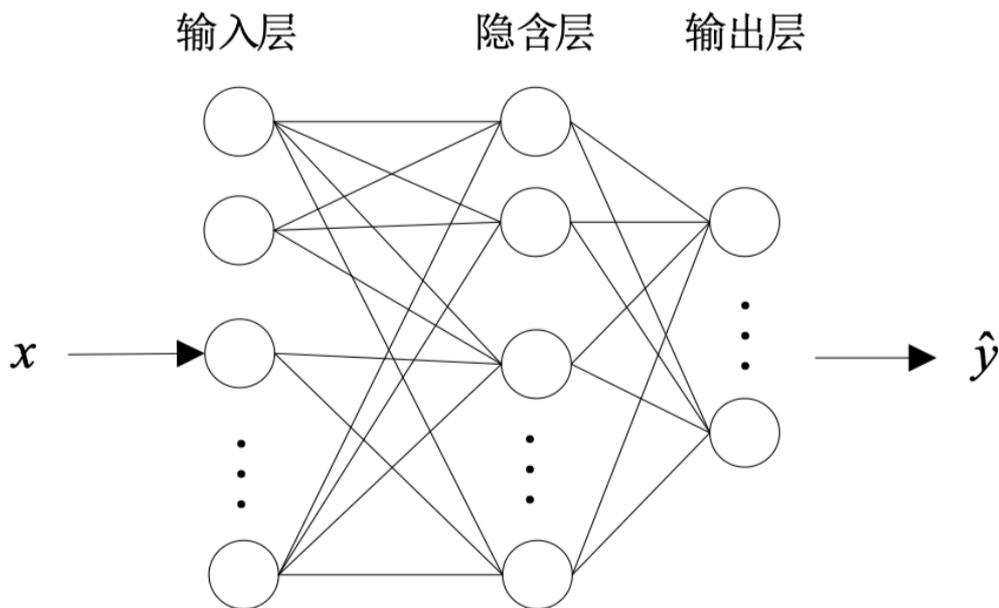


# ChatGPT



# 全连接神经网络的基本原理

- 全连接神经网络包含三层结构：**输入层**、**隐含层（一层或多层）**、**输出层**。所谓全连接，就是**每一层中的神经元与后面一层的各个神经元都有连接**，同层神经元之间不存在连接，也不存在跨层连接。
  - ✓ **输入层**用来接收样本的特征输入；
  - ✓ **隐含层**用于对输入的特征进行加工处理，通过其内部神经元的激活函数实现；
  - ✓ **输出层**用于接收隐含层的加工信息并输出分类结果。

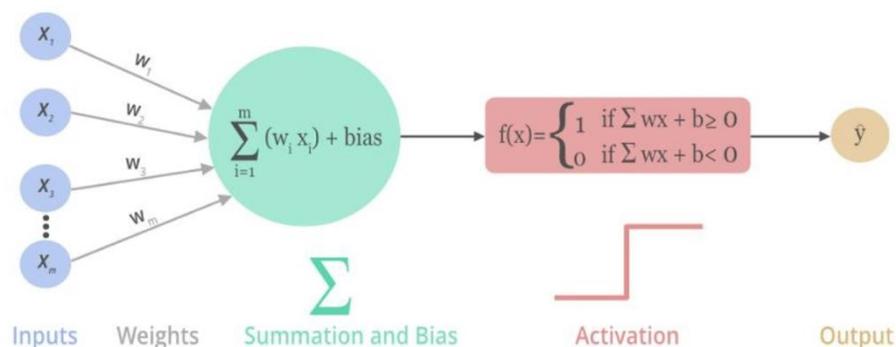


# 全连接神经网络的基本原理

## □ 前向传播 (计算过程)

$$h(x) = g(W_1x + \theta_1)$$

$$\hat{y} = f(W_2h(x) + \theta_2)$$

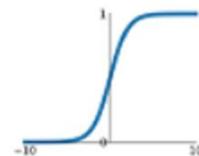


## □ 激活函数: $g, f$ 为**激活函数**, 常用激活函数如下

激活函数	表达式	特点
sigmoid	$S(x) = \frac{1}{1 + e^{-x}}$	将数值映射至 (0, 1) 的区间; 在数值相差不大时效果较好
ReLU	$\text{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$	计算过程简单, 目前最受欢迎的激活函数
Softmax	$S_i = \frac{e^i}{\sum_j e^j}$	将数值映射至 (0, 1) 的区间; 通常用于多分类神经网络输出

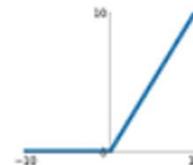
### Sigmoid

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



### ReLU

$$\max(0, x)$$



# 全连接神经网络的基本原理

□ 反向传播 (训练过程)

$$Loss = \frac{1}{2K} \sum_{k=1}^K (\hat{y}^k - y^k)^2$$

(当损失函数为均方误差时)

□ 反向传播基于梯度下降算法，以**目标的负梯度方向**对参数进行更新。

# TensorFlow

- TensorFlow 是由谷歌在 2015 年 11 月发布的深度学习开源工具，可用来快速构建神经网络，并训练深度学习模型。
- TensorFlow的底端由C++实现，计算速度更快。TensorFlow有一个高级机器学习API（tf.contrib.learn），可以更容易地配置、训练和评估大量的机器学习模型。

	PyTorch	TensorFlow
<b>计算图分类</b>	动态计算图	静态计算图
<b>计算图定义</b>	计算图在运行时定义	计算图需提前定义
<b>调试</b>	较简单，可用任何python开发工具（例如：PyCharm）	较复杂，只能用专为Tensorflow开发的工具（例如：tfdbg）
<b>可视化</b>	支持Tensorboard	支持Tensorboard
<b>数据并行</b>	极其简单，只需一行代码	较复杂，需要手动配置
<b>支持硬件</b>	CPU, GPU	CPU, GPU
<b>支持语言</b>	Python, C++	Python, C++
<b>开发公司</b>	Facebook	Google

# TensorFlow

## □ TensorFlow2.0的安装

- ✓ `pip install tensorflow` #安装cpu版
- ✓ `pip install tensorflow-gpu` #安装gpu版
- ✓ `import tensorflow as tf`

# TensorFlow

## □ TensorFlow2.0的张量

- ✓ 与NumPy类似，1 维的数组称之为向量，2 维的数组称之为矩阵，**N 维数组则称之为张量 (Tensor)**。
- ✓ TensorFlow 中，数据都使用张量进行表述。
- ✓ `tf.Variable` : Tensor变量
- ✓ `tf.constant` : Tensor常量
- ✓ 基本运算
- ✓ 自动微分和梯度计算

# TensorFlow

## □ TensorFlow2.0的常用模块

- ✓ tf: 包含了张量定义、变换等常用函数和类;
- ✓ tf.data: 输入数据处理模块, 提供了如 tf.data.Dataset 等类, 用于封装输入数据、指定批量大小等;
- ✓ tf.image: 图像处理模块, 提供了如图像裁剪、变换、编码、解码等类;
- ✓ tf.keras: 原 Keras 框架高阶 API, 包含原 tf.layers 中的高阶神经网络层;
- ✓ tf.linalg: 线性代数模块, 提供了大量线性代数计算方法和类;
- ✓ tf.losses: 损失函数模块, 便于定义神经网络的损失函数;
- ✓ tf.math: 数学计算模块, 提供了大量数学计算函数;
- ✓ tf.saved\_model: 模型保存模块, 可用于模型的保存和恢复;
- ✓ tf.train: 提供用于训练的组件, 例如优化器、学习率衰减策略等;
- ✓ tf.nn: 提供用于构建神经网络的底层函数, 以帮助实现深度神经网络的各类功能层;
- ✓ tf.estimator: 高阶 API, 提供了预创建的 Estimator 或自定义组件。

# Keras

- Keras是由纯Python编写的面向对象的基于TensorFlow的深度学习库。
- Keras包含许多常用神经网络构建块的实现，例如层、目标函数、激活函数、优化器和一系列工具，其代码托管在Github上。
- Keras里有两种搭建神经网络模型的方式，一种是 **Sequential 模型**，另一种是 **Model 模型**。

# Keras

## □ Keras的安装

- ✓ pip install keras
- ✓ import keras

# Keras

## □ Keras的**序列模型**

### 1. 构建模型

```
from keras import models # 导入models模块用于组装各个组件
```

```
from keras import layers # 导入layers模块用于生成神经网络层
```

```
neural_net = models.Sequential() # 组合层级叠加的网络架构
```

```
# 添加模型的网络层
```

```
neural_net.add(layers.Dense(units=64, activation='relu', input_shape=(28*28, )))
```

```
neural_net.add(layers.Dense(units=64, activation='relu'))
```

```
neural_net.add(layers.Dense(units=10, activation='softmax'))
```

```
sequential.summary() # 输出网络初始化的各层的参数状况
```

# Keras

## □ Keras的**序列模型**

### 2. 训练模型

```
neural_net.compile(optimizer= keras.optimizers.SGD(lr = 0.1), loss='mse',  
                   metrics=['accuracy'])      # 配置网络的训练方法  
history = neural_net.fit(x, y, epochs, batch_size) # 执行模型的训练
```

### 3. 测试模型

```
test_loss, test_acc = neural_net.evaluate(data, label, batch_size) # 测试模型
```

# Keras

## □ Keras的函数式模型

### 1. 构建模型

```
from keras.layers import Input, Dense # Input用于接收模型的输入; Dense
                                        # 用于构建全连接层
from keras.models import Model        # 用于构建Model 模型
inputs = Input(shape=(784,))          # 设置模型输入数据的格式
x = Dense(64, activation='relu')(inputs) # 全连接层的输入层
x = Dense(64, activation='relu')(x)     # 全连接层的隐藏层
predictions = Dense(10, activation='softmax')(x) # 全连接层的输出层
model = Model(inputs=inputs, outputs=predictions) # 构建模型并设置模型的
                                                # 输入和输出
```

# Keras

## □ Keras的函数式模型

### 2. 训练模型

# 设置模型的优化器、损失函数、性能指标

```
model.compile(optimizer='rmsprop',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

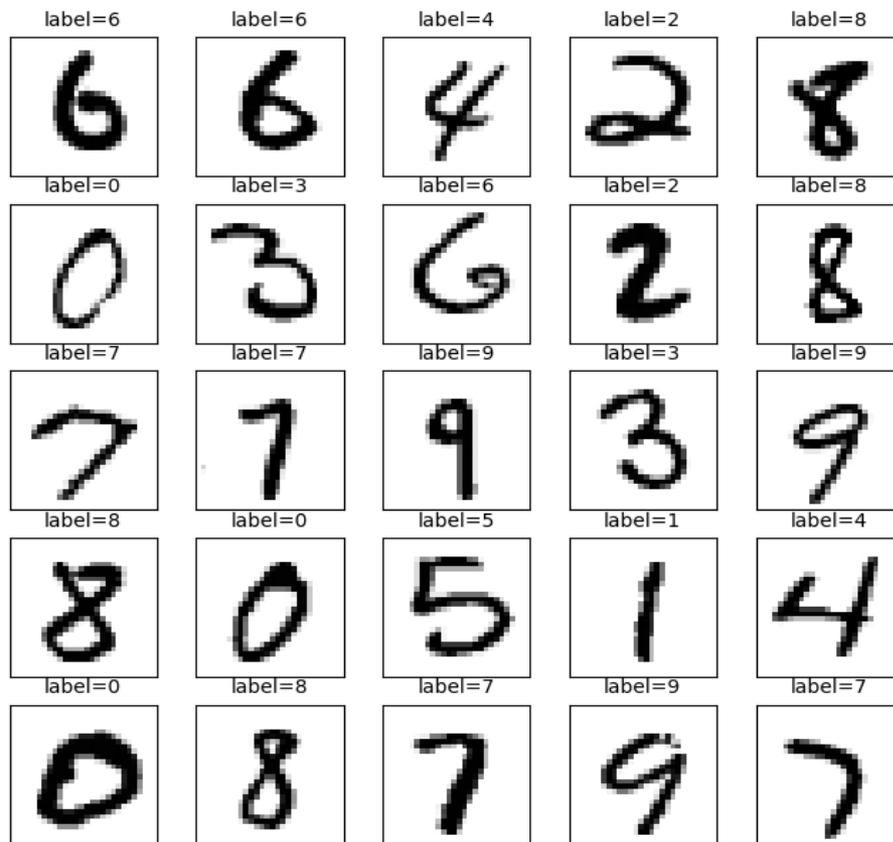
```
model.fit(data, labels) # 训练模型
```

### 3. 测试模型

```
model.predict(self, x, batch_size=32, verbose=0) # 测试模型
```

# 基于Keras库构建全连接神经网络

□ 以手写数字图片数据集MNIST为例，实现对手写数字图片的分类与识别



# 基于Keras库构建全连接神经网络

- 导入所用框架和库
- 获取并处理训练集和测试集
- 构建全连接神经网络

```
# 组合层级叠加的网络架构
```

```
neural_net = models.Sequential()
```

```
# 设置隐含层的神经元 # 数量和激活函数
```

```
neural_net.add(layers.Dense(units=512, activation='relu',  
                             input_shape=(28*28, )))
```

```
# 设置输出层的神经元数量和激活函数
```

```
neural_net.add(layers.Dense(units=10, activation='softmax'))
```

```
# 输出网络初始化的各层的参数状况
```

```
neural_net.summary()
```

# 基于Keras库构建全连接神经网络

# 配置网络的训练方法

```
neural_net.compile(  
    optimizer= tf.keras.optimizers.SGD(lr = 0.1), # 设置优化算法为随机梯度下降,  
                                                    # 学习率为0.1  
    loss='mse', # 设置损失函数为均方误差  
    metrics=['accuracy'] # 设置变量accuracy用于存储分类准确率  
)
```

# 执行模型的训练

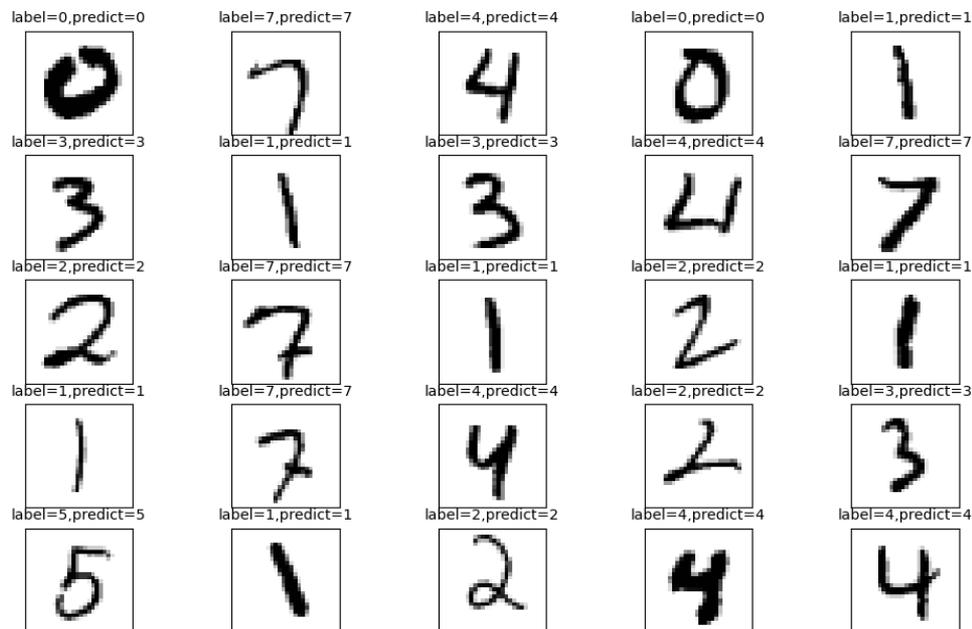
```
history = neural_net.fit(train_images, train_labels, epochs=50, batch_size=512)
```

# 测试训练后的模型

```
test_loss, test_acc = neural_net.evaluate(test_images, test_labels)
```

```
print("Train Epoch:", '%02d' % (epoch + 1), "Loss=", "{:.9f}".format(loss), "  
Accuracy=", acc)
```

# 全连接神经网络实现图片识别



```
Train Epoch: 45 Loss= 0.080445871 Accuracy= 0.8864000082015991
Train Epoch: 46 Loss= 0.080929279 Accuracy= 0.8861999750137329
Train Epoch: 47 Loss= 0.081528306 Accuracy= 0.8860000014305115
Train Epoch: 48 Loss= 0.082433820 Accuracy= 0.8855999946594239
Train Epoch: 49 Loss= 0.083543174 Accuracy= 0.8861999750137329
Train Epoch: 50 Loss= 0.085390568 Accuracy= 0.8858000278472901
acc: 0.87049997
```

# 全连接神经网络的自定义编码实现

## □ 定义主函数

# 在主函数中调用模型的训练和测试函数

```
if __name__ == '__main__':
```

```
    # 训练模型
```

```
    model = train(mnist.train.images, mnist.train.labels, mnist.validation.images,  
                 mnist.validation.labels)
```

```
    # 测试模型并得到最优的模型
```

```
    accuracy = test(model, mnist.test.images, mnist.test.labels)
```

```
    print('test the best model, accuracy=%.2f' % (accuracy)) # 输出准确率
```

# 全连接神经网络的自定义编码实现

- 导入所用框架和库
- 获取并处理训练集、验证集和测试集
- 定义全连接神经网络类

```
class FullConnectionLayer():  
    def __init__(self): # 用于初始化类实例  
        self.mem = {} # 定义在实例化对象中, 以便类全局使用  
  
    def forward(self, X, W): # 前向传播, X为输入数据, W为网络的连接权重矩阵  
        self.mem["X"] = X # 接收输入数据  
        self.mem["W"] = W # 接收网络参数  
        H = np.matmul(X, W) # 计算网络输出  
        return H  
  
    def backward(self, grad_H): # 反向传播, grad_H为损失函数关于H的梯度  
        X = self.mem["X"] # 输入数据  
        W = self.mem["W"]  
        grad_X = np.matmul(grad_H, W.T) # 损失函数关于X的梯度  
        grad_W = np.matmul(X.T, grad_H) # 损失函数关于W的梯度  
        return grad_X, grad_W
```

# 全连接神经网络的自定义编码实现

## □ 定义激活函数和损失函数

- ✓ 定义ReLU类用于隐含层（非线性）
- ✓ 定义交叉熵损失函数类用于训练
- ✓ 定义Softmax类用于输出分类结果

## □ 构建全连接神经网络模型

```
class FullConnectionModel(): # 全连接神经网络模型类
    def __init__(self, latent_dims): # 构造函数
        self.W1 = np.random.normal(loc=0, scale=1, size=[28*28+1, latent_dims])/np.sqrt((28*28+1)/2) # 初始化W1
        self.W2 = np.random.normal(loc=0, scale=1, size=[latent_dims, 10])/np.sqrt(latent_dims/2) # He 初始化W2
        self.mul_h1 = FullConnectionLayer() # 隐含层
        self.relu = Relu() # 激活函数
        self.mul_h2 = FullConnectionLayer() # 输出层
        self.softmax = Softmax() # 激活函数
        self.cross_en = CrossEntropy() # 损失函数
```

# 全连接神经网络的自定义编码实现

```
def forward(self, X, labels): # 前向传播
    bias = np.ones(shape=[X.shape[0], 1]) # 生成权值均为1的矩阵作为偏置
    X = np.concatenate([X, bias], axis=1) # 合并矩阵
    self.h1 = self.mul_h1.forward(X, self.W1) # 隐含层的计算
    self.h1_relu = self.relu.forward(self.h1) # 经过隐含层激活函数处理
    self.h2 = self.mul_h2.forward(self.h1_relu, self.W2) # 输出层的计算
    self.h2_soft = self.softmax.forward(self.h2) # 经过输出层激活函数处理
    self.loss = self.cross_en.forward(self.h2_soft, labels) # 计算交叉熵损失函数
```

```
def backward(self, labels): # 反向传播
    self.loss_grad = self.cross_en.backward(labels) # 计算损失函数在输出层的梯度
    self.h2_soft_grad = self.softmax.backward(self.loss_grad) # 对softmax求导
    self.h2_grad, self.W2_grad = self.mul_h2.backward(self.h2_soft_grad) # 求导
    self.h1_relu_grad = self.relu.backward(self.h2_grad) # 对relu求导
    self.h1_grad, self.W1_grad = self.mul_h1.backward(self.h1_relu_grad) # 计算隐含层的梯度
```

# 全连接神经网络的自定义编码实现

## □ 定义训练函数

- ✓ 定义计算准确率的函数
- ✓ 定义单步训练函数
- ✓ 定义批量训练函数

## □ 定义测试函数

```
def test(model, x, y): # 测试函数
    model.forward(x, y) # 运行模型
    accuracy = computeAccuracy(model.h2_soft, y) # 获取准确率
    return accuracy
```

# 全连接神经网络的自定义编码实现

## □ 程序运行结果:

```
Start seaching the best parameter...

Parameter latent_dims=100, epoch=20 , loss=0.61235563, accuracy=0.798364: 100%|██████████| 20/20 [00:22<00:00, 1.15s/it]
Parameter latent_dims=100, validation_loss=0.5979167190591659, validation_accuracy=0.797.

Parameter latent_dims=200, epoch=20 , loss=0.5519253 , accuracy=0.851945: 100%|██████████| 20/20 [00:25<00:00, 1.28s/it]
Parameter latent_dims=200, validation_loss=0.5074925231099131, validation_accuracy=0.853.

Parameter latent_dims=300, epoch=20 , loss=0.7119555 , accuracy=0.773982: 100%|██████████| 20/20 [00:31<00:00, 1.59s/it]
Parameter latent_dims=300, validation_loss=0.5305907522703492, validation_accuracy=0.842.

The best parameter is 200.

Start training the best model...
Training the best model, epoch=50 , loss=0.31931881, accuracy=0.91 : 100%|██████████| 50/50 [01:07<00:00, 1.35s/it]
Evaluate the best model, test loss=0.30098108, accuracy=0.916000.
```

# 回顾整个过程

## 1. 问题建模

- 回归、分类、聚类 ...

## 2. 收集数据

- 回归：特征数据 $X$ ，连续数据 $y$
- 分类：特征数据 $X$ ，离散数据 $y$

## 3. 特征预处理

- 归一化
- 类别特征
- 时间特征
- 图像数据、序列数据、图结构数据

## 4. 构建模型

模型选择：线性回归、Logistic Regression

KNN、K-Means、神经网络

损失函数：均方误差，交叉熵

## 5. 模型验证&参数调优

使用训练数据训练模型，使用验证数据进行参数调优

验证指标：回归 (wmape、 $R^2$ 、均方误差)

## 6. 模型上线/AB测试

在测试数据上进行模型测试 (测试数据和训练数据来自于同一分布)

# 目录

- 神经网络简介
- TensorFlow
- Keras
- 全连接神经网络及Keras实现
- 全连接神经网络的自定义编码实现
- 卷积神经网络及TensorFlow实现
- AlexNet编码实现

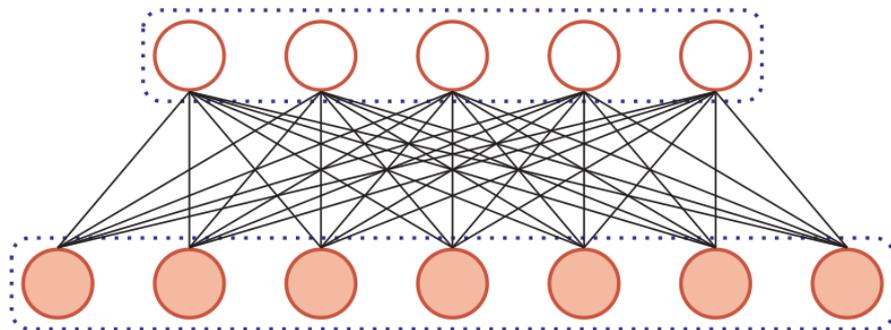
# 图像分类任务

Top5  
错误率



# 全连接神经网络

□ 权重矩阵的参数非常多



□ 局部不变性特征

- ✓ 自然图像中的物体都具有局部不变性特征
  - 尺度缩放、平移、旋转等操作不影响其语义信息
- ✓ 全连接前馈网络很难提取这些局部不变特征

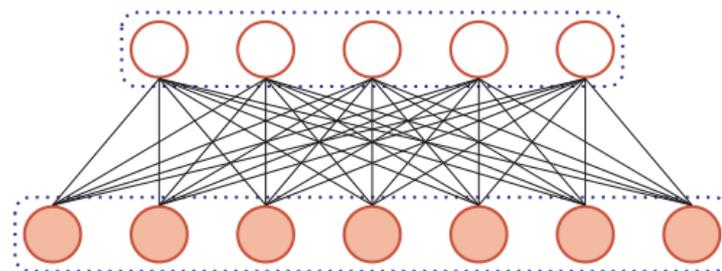
# 卷积神经网络

## □ 卷积神经网络 (Convolutional Neural Networks, CNN)

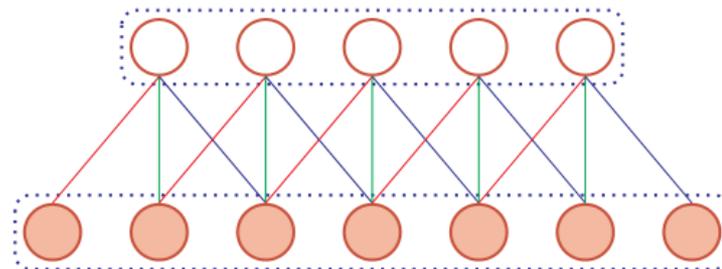
- ✓ 一种前馈神经网络
- ✓ 受生物学上感受野 (Receptive Field) 的机制而提出的
  - 在视觉神经系统中，一个神经元的感受野是指视网膜上的特定区域，只有这个区域内的刺激才能够激活该神经元

## □ 卷积神经网络有三个结构上的特性：

- ✓ 局部连接
- ✓ 权重共享
- ✓ 空间或时间上的次采样



(a) 全连接层



(b) 卷积层

# 卷积

- 卷积经常用在信号处理中，用于计算信号的延迟累积
- 假设一个**信号发生器**每个时刻  $t$  产生一个信号  $x_t$ ，其信息的衰减率为  $w_k$ ，即在  $k-1$  个时间步长后，信息为原来的  $w_k$  倍

✓ 假设  $w_1 = 1, w_2 = 1/2, w_3 = 1/4$

- 时刻  $t$  收到的信号  $y_t$  为当前时刻产生的信息和以前时刻延迟信息的叠加

$$y_t = 1 \times x_t + 1/2 \times x_{t-1} + 1/4 \times x_{t-2}$$

$$= w_1 \times x_t + w_2 \times x_{t-1} + w_3 \times x_{t-2}$$

$$= \sum_{k=1}^3 w_k \cdot x_{t-k+1}$$

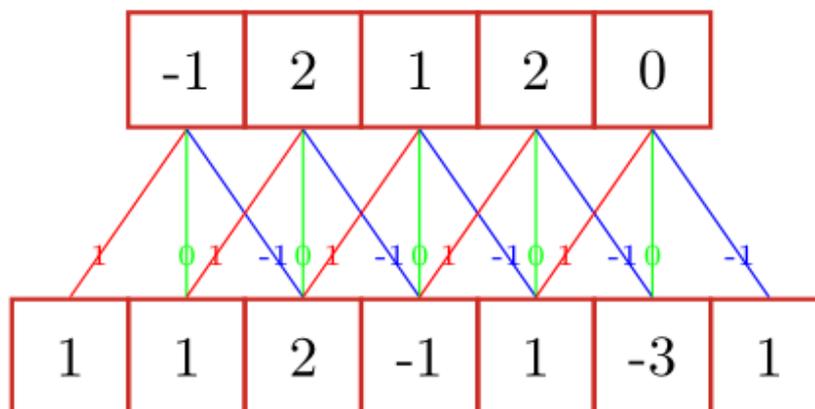
滤波器 (filter) 或卷积核 (convolution kernel)

# 卷积

□ 给定一个输入信号序列  $x$  和滤波器  $w$ , 卷积的输出为:

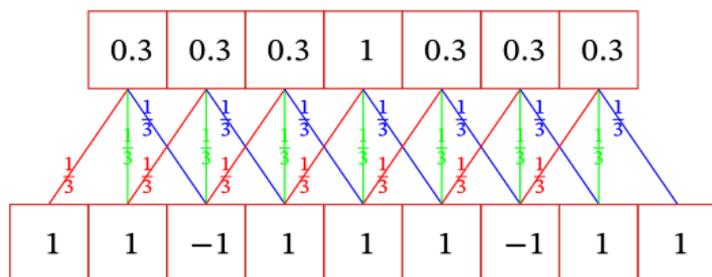
$$y_t = \sum_{k=1}^K w_k x_{t-k+1}$$

Filter: [-1,0,1]



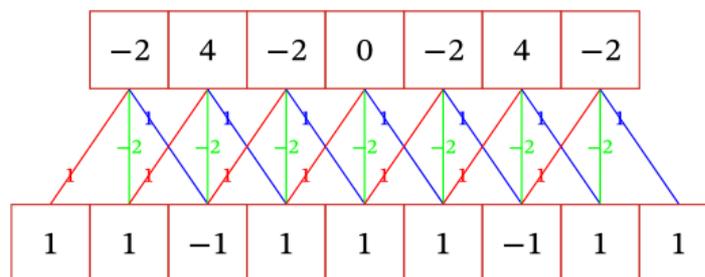
# 卷积

□ 不同的滤波器来提取信号序列中的不同特征



(a) 滤波器  $[1/3, 1/3, 1/3]$

低频信息



(b) 滤波器  $[1, -2, 1]$

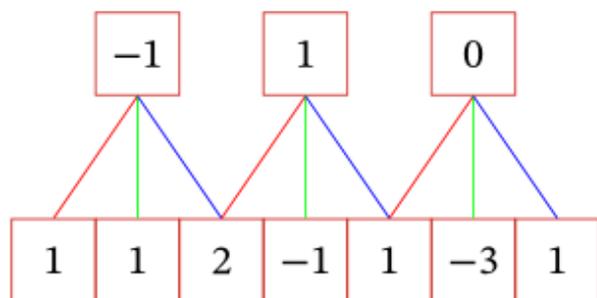
高频信息

$$y''(u) = y(u + 1) + y(u - 1) - 2y(u)$$

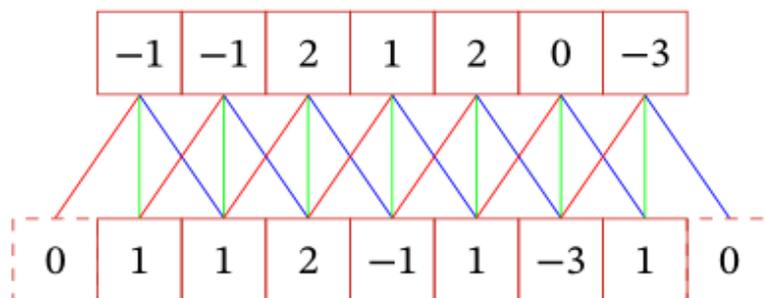
二阶微分

# 卷积扩展

□ 引入滤波器的滑动步长  $S$  和零填充  $P$



(a) 步长  $S = 2$



(b) 零填充  $P = 1$

# 卷积类型

□ 输入神经元个数为  $M$ ，卷积大小为  $K$ ，卷积的结果按输出长度不同可以分为三类：

✓ 窄卷积：步长  $S=1$ ，两端不补零  $P=0$ ，卷积后输出长度为  $M-K+1$

✓ 宽卷积：步长  $S=1$ ，两端补零  $P=K-1$ ，卷积后输出长度为  $M+K-1$

✓ 等宽卷积：步长  $S=1$ ，两端补零  $P=(K-1)/2$ ，卷积后输出长度为  $M$

□ 在早期的文献中，卷积一般默认为窄卷积

□ 而目前的文献中，卷积一般默认为等宽卷积

# 二维卷积

□ 在图像处理中，图像是以二维矩阵的形式输入到神经网络中，因此我们需要二维卷积

□ 一个输入信息  $X$  和滤波器  $W$  的二维卷积定义为  $Y = W * X$

$$y_{ij} = \sum_{u=1}^U \sum_{v=1}^V w_{uv} x_{i-u+1, j-v+1}$$

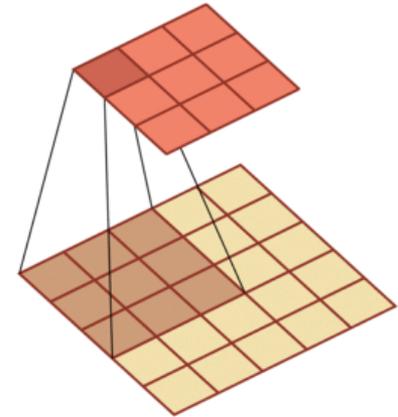
1	1	1	1	1
-1	0	-3	0	1
2	1	1	-1	0
0	-1	1	2	1
1	2	1	1	1

\*

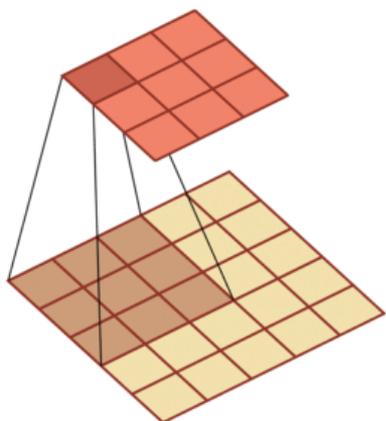
-1	0	0
0	0	0
0	0	1

=

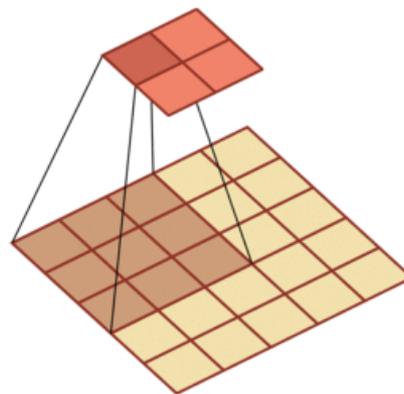
0	-2	-1
2	2	4
-1	0	0



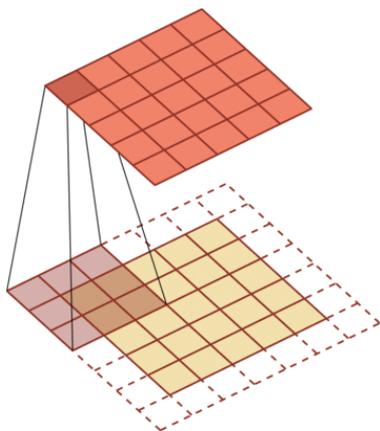
# 二维卷积



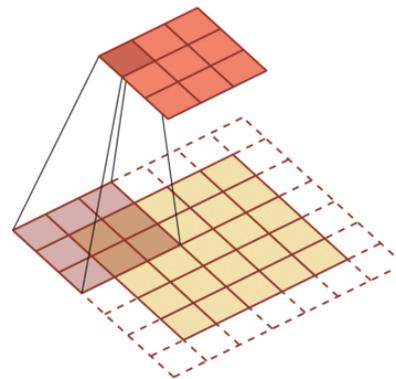
步长1, 零填充0



步长2, 零填充0

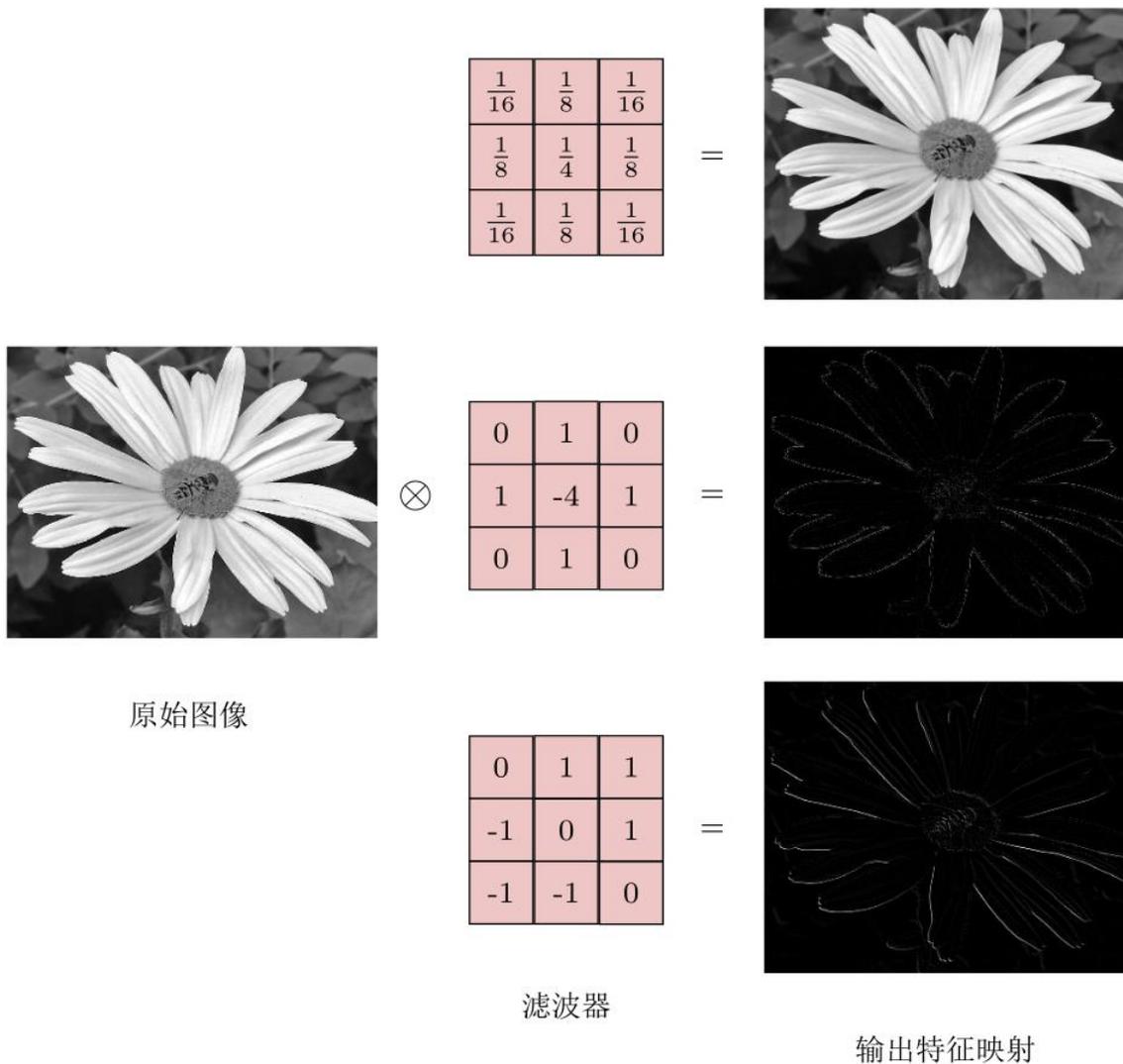


步长1, 零填充1



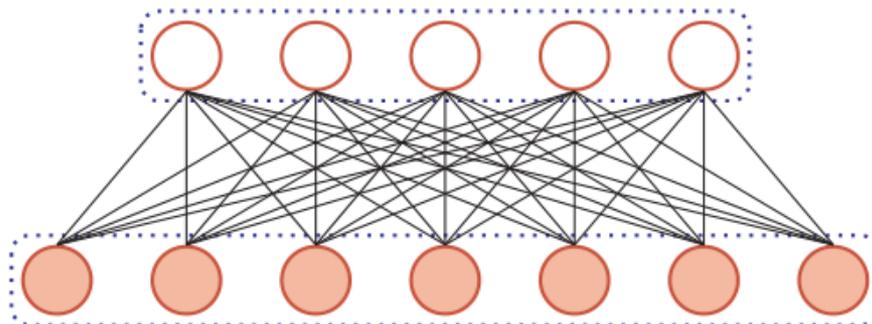
步长2, 零填充1

# 卷积作为特征提取器

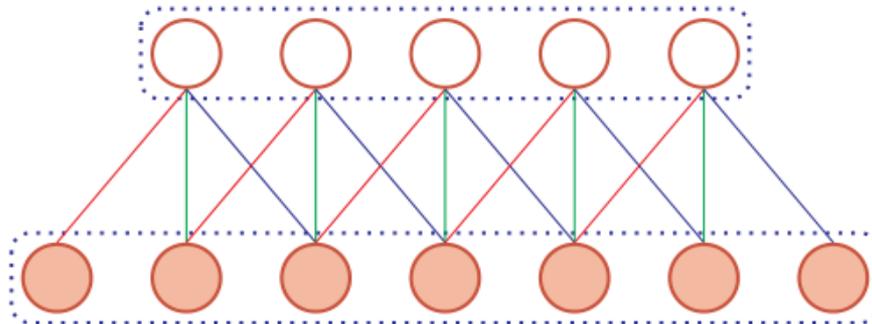


# 卷积神经网络

□ 用卷积层代替全连接层



(a) 全连接层

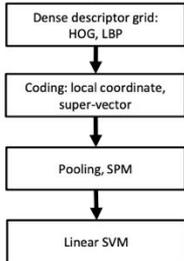


(b) 卷积层

# 图像分类任务

## Year 2010

### NEC-UIUC

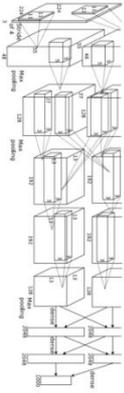


[Lin CVPR 2011]

Lion image by Swissfrog is licensed under CC BY SA

## Year 2012

### SuperVision

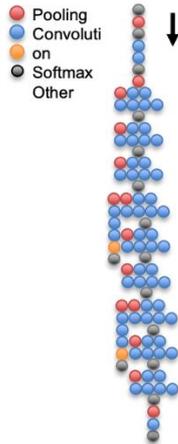


[Krizhevsky NIPS 2012]

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

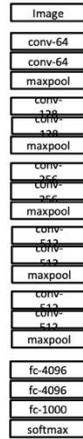
## Year 2014

### GoogLeNet



[Szegedy arxiv 2014]

### VGG



[Simonyan arxiv 2014]

## Year 2015

### MSRA

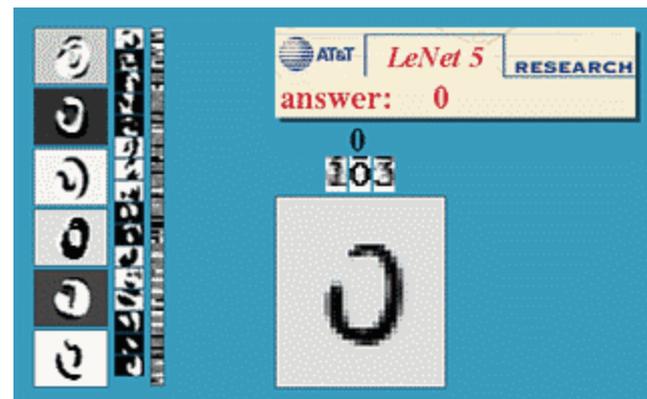
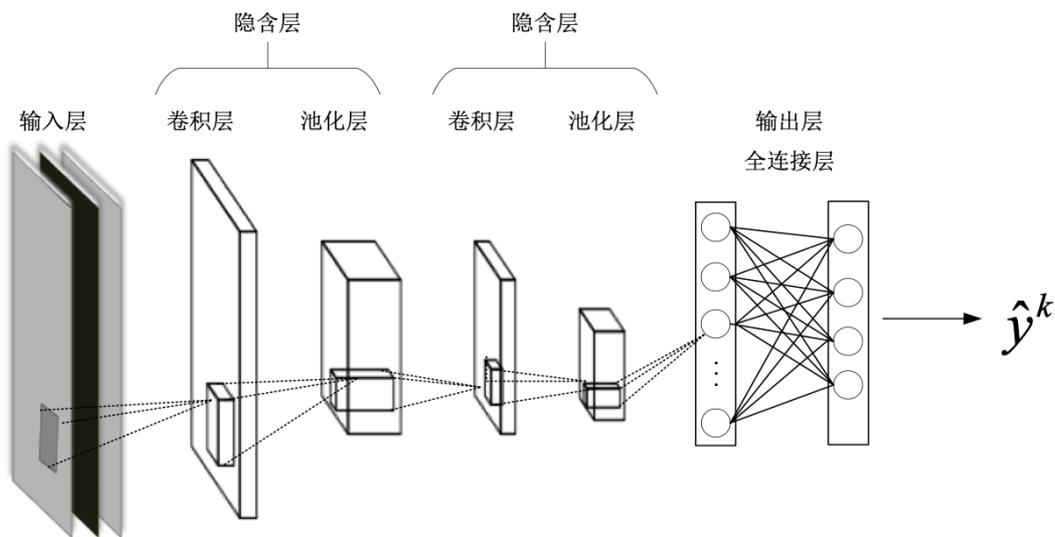


[He ICCV 2015]

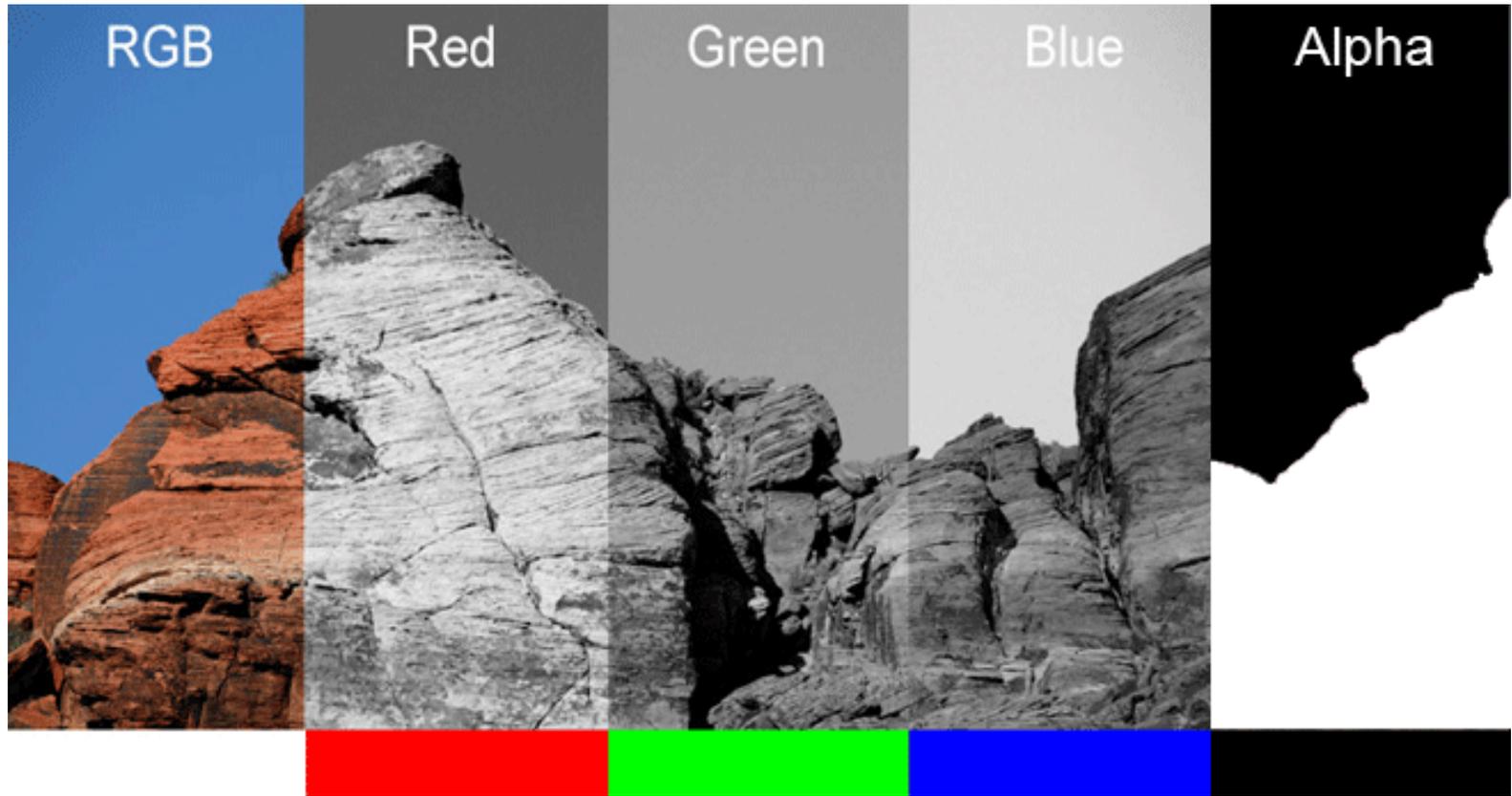
	参数量	深度
Alexnet	$6.1 \times 10^7$	7
VGG16	$1.38 \times 10^8$	16
Googlenet	$1.3 \times 10^7$	22
Resnet50	$2.5 \times 10^7$	50
Resnet152	$6.02 \times 10^7$	152

# 卷积神经网络基本原理

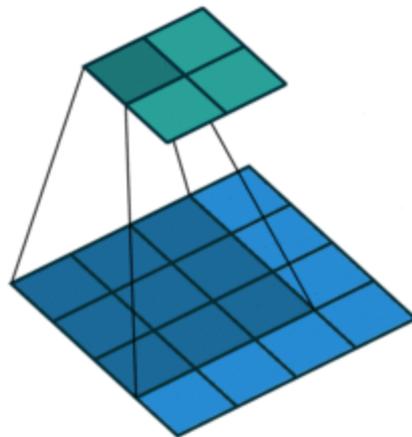
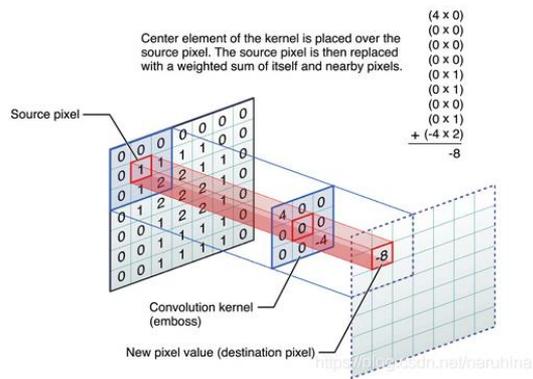
- 卷积神经网络包含三层结构：**输入层、隐含层（一层或多层）和输出层。**卷积神经网络的**输入层**可以处理**多维数据**，隐含层可包含**卷积层**和**池化层**。输入层用来接收样本的特征输入；
- 卷积层对输入数据进行特征提取，**其内部可以有多个不同的卷积核。**池化层通过**特征选择简化在卷积层提取到的特征，减少无效特征。**



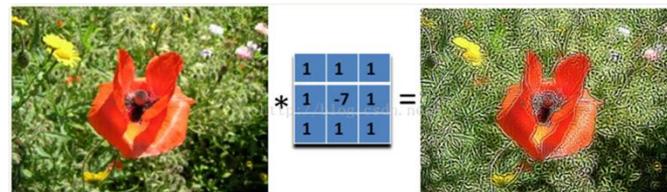
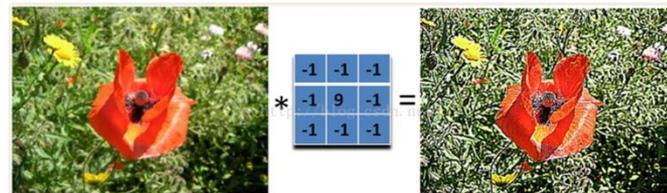
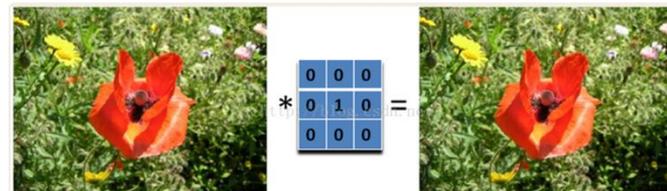
# 输入层



# 卷积层



## 三种图像卷积

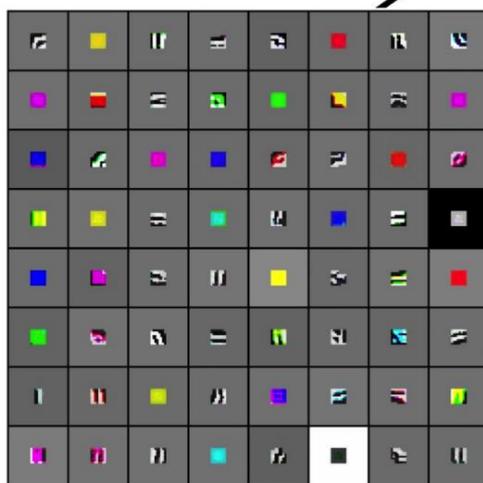


# 卷积层

## Preview

[Zeiler and Fergus 2013]

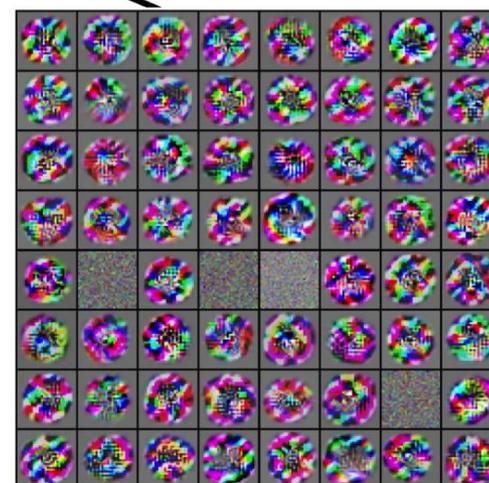
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014]



VGG-16 Conv1\_1



VGG-16 Conv3\_2



VGG-16 Conv5\_3

# 池化层

Max Pooling

29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2  
pool size

100	184
12	45

Average Pooling

31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2  
pool size

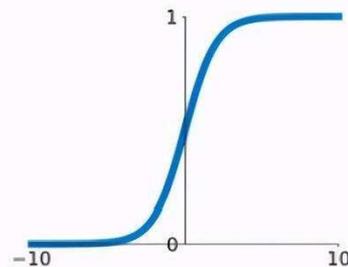
36	80
12	15

<https://blog.csdn.net/QFJIZHI>

# 激活层

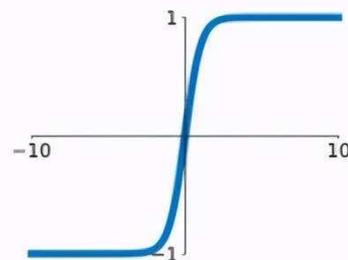
**Sigmoid**

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



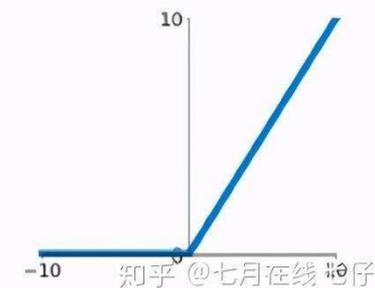
**tanh**

$$\tanh(x)$$



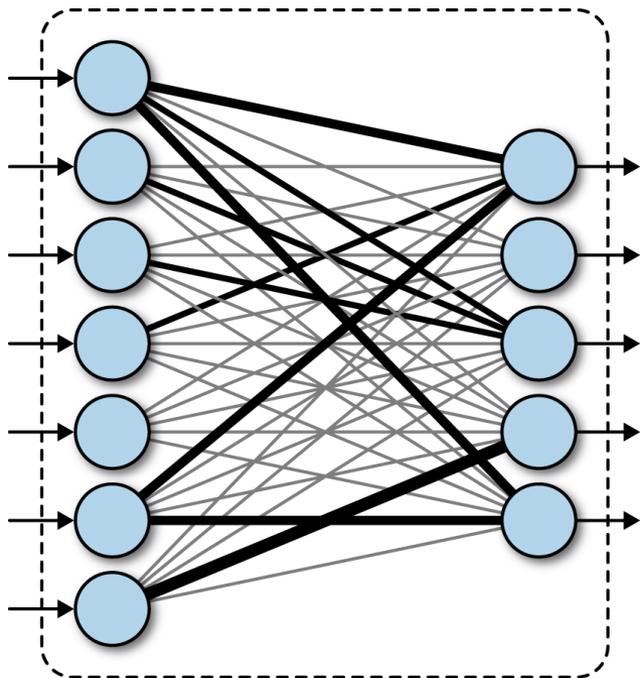
**ReLU**

$$\max(0, x)$$



# 输出层

全连接层



softmax  
(如果是分类任务)

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

# 卷积神经网络基本原理

## □ 前向传播(计算过程)

- ✓ 第一个卷积层

$$z_{u,v}^1 = \sum_{i=1}^5 \sum_{j=1}^5 x_{i+u-1,j+v-1} \cdot k_{i,j}^1 + b^1, u, v \in [1, 28]$$

- ✓ 第一个池化层  $A^1(14, 14) = [a_{i,j}^1, i, j \in [1, 14]]$

$$a_{i,j}^1 = p(z_{u-1:u+1,v-1:v+1}^1)(u, v = 2, 4, \dots, 26, 28)$$

- ✓ 第二个卷积层和第二个池化层同上，第二个池化层的输出为  $A^2(7, 7)$
- ✓ 全连接层进行分类

$$h(A^2) = g(W_1 A^2(7, 7) + \theta_1)$$

$$\hat{y} = \text{softmax}(h)$$

# 卷积神经网络基本原理

## □ 反向传播(训练过程)

- ✓ 和全连接神经网络相比，卷积神经网络的训练要更加复杂，但训练原理都是一样的。
- ✓ 将交叉熵损失作为损失函数

$$CrossEntLoss = \sum_{k=1}^K \hat{y}^k \ln y^k$$

- ✓ 链式法则求导计算损失函数对参数的梯度，根据梯度下降算法更新待训练参数

# 基于Keras库构建全连接神经网络

1. 导入所用框架和库
2. 获取并处理训练集和测试集
3. 构建卷积神经网络模型

```
model = Sequential() # Sequential模型
```

```
model.add(Conv2D(filters=8, kernel_size=(5, 5), padding='same', input_shape=(28, 28, 1), activation='relu')) # 第一个卷积层
```

```
model.add(MaxPooling2D(pool_size=(2, 2))) # 第一个池化层
```

```
model.add(Conv2D(filters=16, kernel_size=(5, 5), padding='same', activation='relu')) # 第二个卷积层
```

```
model.add(MaxPooling2D(pool_size=(2, 2))) # 第二个池化层
```

```
model.add(Flatten()) # 将第二个池化层的输出扁平化为一维数据
```

```
model.add(Dense(100, activation='relu')) # 全连接层的隐含层
```

```
model.add(Dropout(0.25)) # 用来放弃一些权值，防止过拟合
```

```
model.add(Dense(10, activation='softmax')) # 全连接层的输出层
```

# 基于Keras库构建全连接神经网络

1. 导入所用框架和库
2. 获取并处理训练集和测试集
3. 构建卷积神经网络模型

#compile()用于配置模型的训练方法

```
model.compile(loss='categorical_crossentropy', optimizer='adam',  
              metrics=['accuracy'])
```

#fit()用于执行模型的训练

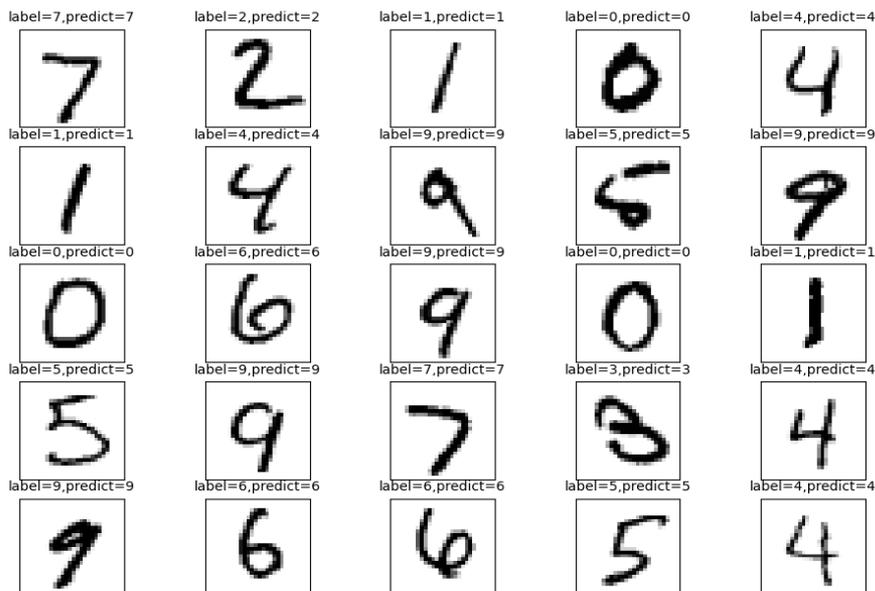
```
history = model.fit(x=x_train, y=y_train, validation_split=0.2, epochs=10,  
                   batch_size=128)
```

#evaluate()用于评估训练后的模型

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

# 卷积神经网络实现图片识别

```
Epoch 5/10
240/240 - 11s - loss: 0.0543 - accuracy: 0.9835 - val_loss: 0.0441 - val_accuracy: 0.9876 - 11s/epoch - 45ms/step
Epoch 6/10
240/240 - 11s - loss: 0.0474 - accuracy: 0.9854 - val_loss: 0.0422 - val_accuracy: 0.9877 - 11s/epoch - 45ms/step
Epoch 7/10
240/240 - 11s - loss: 0.0408 - accuracy: 0.9869 - val_loss: 0.0410 - val_accuracy: 0.9879 - 11s/epoch - 44ms/step
Epoch 8/10
240/240 - 11s - loss: 0.0366 - accuracy: 0.9888 - val_loss: 0.0395 - val_accuracy: 0.9892 - 11s/epoch - 44ms/step
Epoch 9/10
240/240 - 11s - loss: 0.0324 - accuracy: 0.9899 - val_loss: 0.0401 - val_accuracy: 0.9880 - 11s/epoch - 44ms/step
Epoch 10/10
240/240 - 11s - loss: 0.0291 - accuracy: 0.9903 - val_loss: 0.0414 - val_accuracy: 0.9874 - 11s/epoch - 44ms/step
313/313 - 2s - loss: 0.0295 - accuracy: 0.9898 - 2s/epoch - 5ms/step
test loss - 0.03 - test accuracy - 0.99
```



# AlexNet编码实现

## □ AlexNet的特点:

- ✓ 成功使用ReLU作为CNN的激活函数，解决了Sigmoid在网络较深时的梯度弥散问题；
- ✓ 训练时使用Dropout随机忽略一部分神经元，以避免模型过拟合；
- ✓ 在CNN中使用重叠的最大池化；
- ✓ 使用CUDA加速深度卷积网络的训练，利用GPU强大的并行计算能力，处理神经网络训练时大量的矩阵运算；
- ✓ 数据增强。

<https://github.com/uestcsongtaoli/AlexNet>

# AlexNet编码实现

## □ 导入所用框架和库

## □ 其他准备工作

```
tf.random.set_seed(0) # 设置全局随机种子  
batch_size = 128 # 训练批次的样本量  
optimizer = optimizers.Adam(0.000005) # 设置优化器和学习率  
epochs = 2 # 在训练过程中所有训练集反复训练的次数
```

## □ 定义预处理数据的函数

```
def preprocess(x, y): # 对数据集预处理  
    x = tf.cast(x, dtype=tf.float32) / 255. # 对数据归一化  
    y = tf.cast(y, dtype=tf.int32) # 处理标签格式  
    y = tf.one_hot(y, depth=10) # 独热编码  
    return x, y
```

# AlexNet编码实现

## □ 定义AlexNet类

### ✓ 定义卷积层

# 第一个卷积层

```
layers.Conv2D(16, (3, 3), padding='same', strides=1, activation=tf.nn.relu)
```

# 第一个池化层

```
layers.MaxPool2D(pool_size=(2, 2), strides=2, padding='same' )
```

# 第二个卷积层

```
layers.Conv2D(32, (3, 3), padding='same', strides=1, activation=tf.nn.relu)
```

# 第二个池化层

```
layers.MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
```

# 第三个卷积层

```
layers.Conv2D(64, (3, 3), padding='same', strides=1, activation=tf.nn.relu)
```

# 第四个卷积层

```
layers.Conv2D(128, (3, 3), padding='same', strides=1, activation=tf.nn.relu)
```

# 第五个卷积层

```
layers.Conv2D(256, (3, 3), padding='same', strides=1, activation=tf.nn.relu)
```

# 第三个池化层

```
layers.MaxPool2D(pool_size=(2, 2), strides=2, padding='same')
```

# AlexNet编码实现

## □ 定义AlexNet类

### ✓ 定义全连接层

#定义全连接层

```
self.fc = Sequential([ # 全连接
    layers.Dense(4096, activation=tf.nn.relu), # 全连接神经网络的输入层
    layers.Dropout(0.4), # 防止过拟合
    layers.Dense(2048, activation=tf.nn.relu), # 第一个全连接层
    layers.Dropout(0.4), # 防止过拟合
    layers.Dense(1024, activation=tf.nn.relu), # 第二个全连接层
    layers.Dropout(0.4), # 防止过拟合
    layers.Dense(10, activation='softmax') # 全连接神经网络的输出层
])
```

# AlexNet编码实现

## □ 定义主函数

```
if __name__ == '__main__': # 主函数
    # 编译模型
    model.compile(optimizer=optimizers.Adam(learning_rate=0.0001),
                  loss=tf.losses.CategoricalCrossentropy,
                  metrics=['acc' ])

    # 训练模型
    model.fit(train_db, epochs=epochs, validation_data=test_db, validation_freq=2)
    # 保存模型至指定路径
    model.save_weights('./checkpoint/weights.ckpt')
    # 测试模型
    model.evaluate(test_db)
```

# AlexNet编码实现

□ 程序运行结果:

```
Epoch 1/2  
469/469 [=====] - 226s 478ms/step - loss: 0.3387 - acc: 0.8927  
Epoch 2/2  
469/469 [=====] - 231s 491ms/step - loss: 0.0607 - acc: 0.9815 - val_loss: 0.0410 - val_acc: 0.9875  
79/79 [=====] - 7s 84ms/step - loss: 0.0410 - acc: 0.9875
```

# 总结

- TensorFlow基本介绍
- Keras基本介绍
  - ✓ Keras序列模型
  - ✓ Keras函数式模型
- 全连接神经网络、卷积神经网络
  - ✓ 基本原理
  - ✓ 构建过程
  - ✓ 训练过程
  - ✓ 手写数字图片数据集MNIST上的应用
  
- 作业: Spoc第九章 (DDL: 见课程主页)